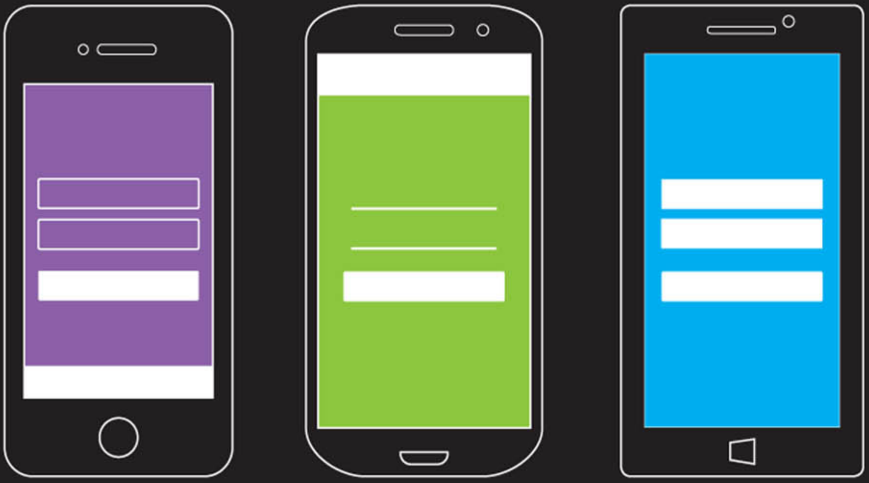
Xamarin

Microsoft

Creating Mobile Apps with Xamarin.Forms



PREVIEW EDITION

Cross-platform C# programming for iOS, Android, and Windows Phone

CHARLES PETZOLD

PREVIEW EDITION

This excerpt provides early content from a book currently in development and is still in draft format. See additional notice below.

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

ISBN: 978-0-7356-9725-6

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions and Project Editor: Devon Musgrave
Cover illustration: Serena Zhang

Table of contents

Introduction

This is a Preview Edition of a book about writing applications for Xamarin.Forms, the exciting new mobile development platform for iOS, Android, and Windows Phone unveiled by Xamarin in May 2014. Xamarin.Forms lets you write shared user-interface code in C# and XAML (the eXtensible Application Markup Language) that maps to native controls on these three platforms.

This book is a Preview Edition because it's not complete. It has only six chapters. We anticipate that the final version of the book will have at least half a dozen additional chapters and that the chapters in this Preview Edition might be fleshed out, enhanced, or completely reconceived. The final edition of the book will probably be published in the spring of 2015.

Who should read this book

This book is for C# programmers who want to write applications for the three most popular mobile platforms: iOS, Android, and Windows Phone with a single code base. Xamarin.Forms also has applicability for those programmers who want eventually to use C# and the Xamarin.iOS and Xamarin.Android libraries to target the native application programming interfaces (APIs) of these platforms. Xamarin.Forms can be a big help in getting started with these platforms or in constructing a prototype or proof-of-concept application.

Assumptions

This book assumes that you know C# and have some familiarity with the use of the .NET Framework. However, when I discuss some C# and .NET features that might be somewhat new to recent C# programmers, I adopt a somewhat slower pace. In particular, the introduction of the `async` keyword and `await` operator in Chapter 3 follows a discussion that shows how to do asynchronous programming using traditional callback methods.

Organization of this book

This book is intended as a tutorial to learn Xamarin.Forms programming. It is not a replacement for the online API documentation, which can be found here under the heading Xamarin.Forms Framework on this page: http://api.xamarin.com/.

This Preview Edition's Chapter 1 discusses Xamarin.Forms in the larger context of mobile development and the Xamarin platform and also covers the hardware and software configurations you'll need. Chapter 2 explores some of the basics of Xamarin.Forms programming, including the use of `Label`, `Button`, and `StackLayout`.

In Chapters 3 and 4, however, I tried to do something a little different: These chapters show the progressive step-by-step development of a small Xamarin.Forms application. Despite the simplicity of this program, it is in many ways a "real" application, and requires essential real-app facilities such as file I/O and application lifecycle handling, both of which turned out to be somewhat more challenging than I originally anticipated. I'm curious to hear whether these two chapters "work" or not. See the section below on submitting feedback to us.

Chapters 5 and 6 return to more conventional API tutorials. My biggest regret is that I wasn't able to get some coverage of XAML into this Preview Edition. However, the Xamarin website has some additional resources for learning Xamarin.Forms including a six-part "XAML for Xamarin.Forms" document: http://developer.xamarin.com/guides/cross-platform/xamarin-forms/.

Conventions and features in this book

This book has just a few typographical conventions:

- All programming elements referenced in the text—including classes, methods, properties, variable names, etc.—are shown in a monospaced font, such as the `StackLayout` class.
- Items that appear in the user interface of Visual Studio or Xamarin Studio, or the applications discussed in these chapters, appear in boldface, such as the **Add New Project** dialog.
- Application solutions and projects also appear in boldface, such as **ColorScroll**.

System requirements

This book assumes that you'll be using Xamarin.Forms to write applications that simultaneously target all three supported mobile platforms—iOS, Android, and Windows Phone. However, it's very likely that many readers will be targeting only one or two platforms in their Xamarin.Forms solutions. The platforms you target—and the Xamarin Platform package you purchase—govern your hardware and software requirements. For targeting iOS devices, you'll need a Mac installed with Apple XCode as well as the Xamarin Platform, which includes Xamarin Studio. For targeting Windows Phone, you'll need Visual Studio 2012 or 2013 (not an Express Edition) on a PC, and you'll need to have installed the Xamarin Platform.

However, you can also use Visual Studio on the PC to target iOS devices if the Mac with XCode and the Xamarin Platform is accessible via WiFi. You can target Android devices from Visual Studio on the PC or from Xamarin Studio on either the PC or Mac.

Chapter 1 has more details on the various configurations you can use, and resources for additional information and support. My setup for creating this book consisted of a Microsoft Surface Pro 2 (with external monitor, keyboard and mouse) installed with Visual Studio 2013 and the Xamarin Platform, connected by WiFi with a MacBook Pro installed with XCode and the Xamarin Platform.

Downloads: Code samples

The sample programs shown in the pages of this book were compiled in early September with version 1.2.2.6243 of Xamarin.Forms. The source code of these samples is hosted on a repository on GitHub: https://github.com/xamarin/xamarin-forms-book-preview/.

You can clone the directory structure to a local drive on your machine, or download a big ZIP file. I'll try to keep the code updated with the latest release of Xamarin.Forms and to fix (and comment) any errors that might have sneaked through.

You can report problems, bugs, or other kinds of feedback about the book or source code by clicking the **Issues** button on this GitHub page. You can search through existing issues or file a new one. To file a new issue, you'll need to join GitHub (if you haven't already).

Use this GibHub page only for issues involving the book. For questions or discussions about Xamarin.Forms itself, use the Xamarin.Forms forum: http://forums.xamarin.com/categories/xamarin-forms.

Updating the code samples

The libraries that comprise Xamarin.Forms are distributed via the NuGet package manager. The Xamarin.Forms package consists of five dynamic-link libraries: Xamarin.Forms.Core.dll, Xamarin.Forms.Xaml.dll, Xamarin.Forms.Platform.iOS.dll, Xamarin.Forms.Platform.Android.dll, and Xamarin.Forms.Platform.WP8.dll. The Xamarin.Forms package also requires Xamarin Support Library v4 (Xamarin.Android.Support.v4.dll) and the Windows Phone Toolkit (Microsoft.Phone.Controls.-Toolkit.dll), which should be automatically included.

When you create a new Xamarin.Forms solution using Visual Studio or Xamarin Studio, a version of the Xamarin.Forms package becomes part of that solution. However, that might not be the latest Xamarin.Forms package available from NuGet. Here's how to update that package:

In Visual Studio, right-click the solution name in the **Solution Explorer** and select **Manage NuGet Packages for Solution**. Select **Installed packages** at the left of the dialog to see what's currently installed, and **Updates** and **nuget.org** at the left to choose to update the package. If an update is available, clicking the **Update All** button is easiest to get it into the solution.

In Xamarin Studio, in the individual projects in the **Solution** list, under **Packages**, select the Xamarin.Forms package and select **Update** from the tool dropdown.

The source code for this book that is stored on GitHub does not include the actual NuGet packages. Xamarin Studio will automatically download them when you load the solution, but Visual Studio will not. After you first load the solution into Visual Studio, right-click the solution name in the **Solution**

Explorer and select **Manage NuGet Packages for Solution**. You should be given the option to restore the packages with a **Restore** button at the top of the dialog. You can then update the package by selecting **Updates** and **nuget.org** at the left and (if an update exists) pressing the **Update All** button.

Acknowledgments

It's always seemed peculiar to me that authors of programming books (such as this one) are sometimes better known to programmers than the people who actually created the product that is the subject of the book! The real brains behind Xamarin.Forms are Jason Smith, Eric Maupin, Stephane Delcroix, and Seth Rosetter. Congratulations, guys! We've been enjoying the fruits of your labor!

Over the months that this Preview Edition was in progress, I have benefited from valuable feedback, corrections, and edits from several people. This book wouldn't exist without the collaboration of Bryan Costanich at Xamarin and Devon Musgrave at Microsoft Press. Both Bryan and Craig Dunn at Xamarin read some of my drafts of early chapters and easily persuaded me to take a somewhat different approach to the material. Later on, Craig kept me on track. During the final days before my deadline for this Preview Edition, Stephane Delcroix offered essential technical reads and John Meade performed copyediting under extreme crunch conditions. Microsoft Press supplemented that with another very helpful technical read by Andy Wigley, who persistently prodded me to make the book better.

Almost nothing I do these days would be possible with the daily companionship and support of my wife, Deirdre Sinnott.

Free ebooks from Microsoft Press

We want to hear from you

CHAPTER 3

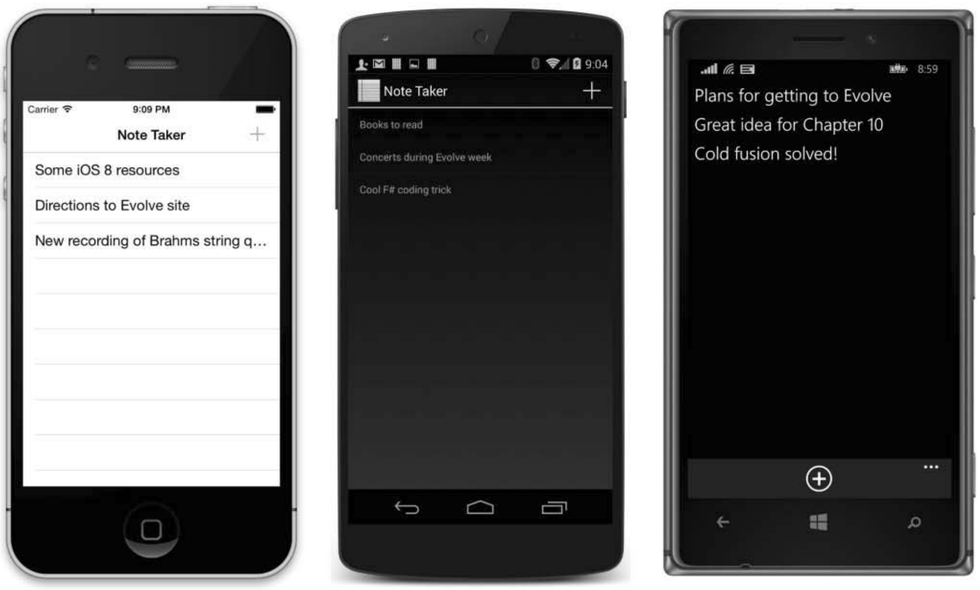
Building an app: Infrastructure

Sometimes programming tutorials such as this one can cover a lot of ground before directly addressing the needs of real-world applications. This chapter and the next are an attempt to avoid that common problem.

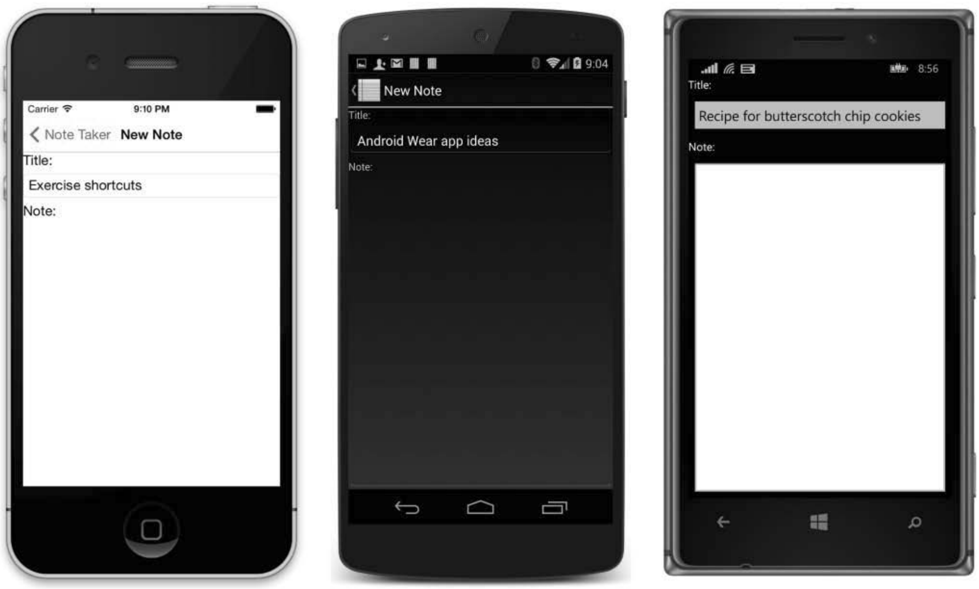
These two chapters present an actual application built from the ground up. This won't be a very sophisticated application, but it will be usable and useful. The application's name is **NoteTaker**, and it lets you take notes and save them.

Each note is some text—as much text as you want—and an optional title. If you don't specify a title, one will be generated for you from the first few words of the note.

When the program is finally finished at the end of the next chapter, it will have the following home screen:


Carrier
9:09 PM
Note Taker
Some iOS 8 resources
Directions to Evolve site
New recording of Brahms string q...
9:04
Note Taker
Books to read
Concerts during Evolve week
Cool F# coding trick
8:59
Plans for getting to Evolve
Great idea for Chapter 10
Cold fusion solved!

That's a scrollable lists of the titles of the notes you've already created. A **New** button symbolized by a plus sign appears as a toolbar item either at the top or bottom of the screen. Tap that button and the program navigates to a page for entering a new note:

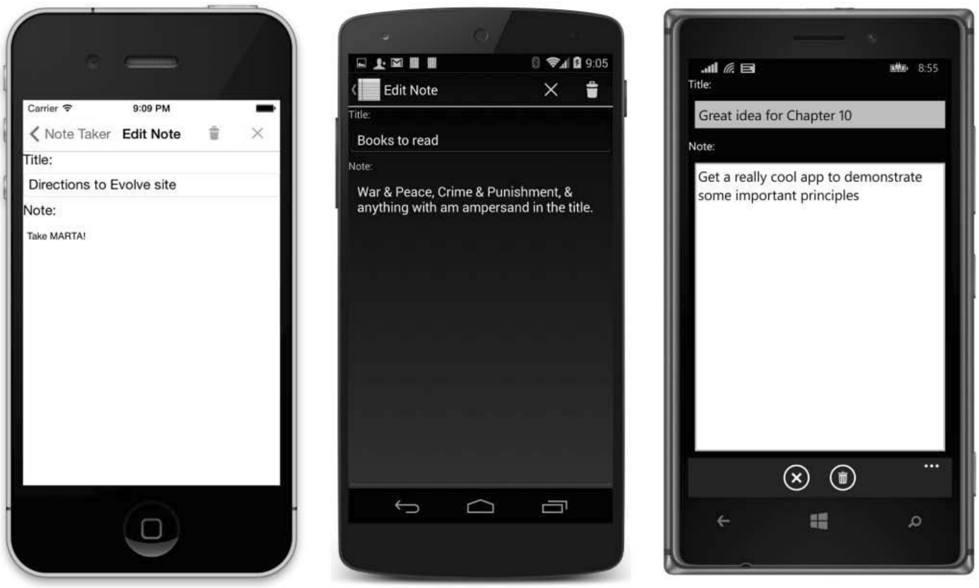


Title:
Exercise shortcuts
Note:

Title:
Android Wear app ideas
Note:

Title:
Recipe for butterscotch chip cookies
Note:

You can also edit an existing note by tapping that note on the home screen:



Title:
Directions to Evolve site
Note:
Take MARTA!

Title:
Books to read
Note:
War & Peace, Crime & Punishment, & anything with am ampersand in the title.

Title:
Great idea for Chapter 10
Note:
Get a really cool app to demonstrate some important principles

Toolbar items allow you to cancel the editing or delete the note. The **New Note** and **Edit Note** screens are very similar, so it's probably not surprising that they are simply variations of the same page.

Engineering in general is a problem-solving activity, and programming is no different. We begin with a vision and then plan a strategy and set some goals to begin realizing that vision in code. But in attempting to build the program, problems and obstacles are often encountered. It is in solving these problems that we battle through our ignorance to acquire knowledge and skills.

In this chapter the **NoteTaker** program goes through six different versions. They are numbered 1 through 6, but the version numbers should probably be more like 0.1 through 0.6. In the course of enhancing this program through these six versions, the following topics are encountered:

- The `Entry` and `Editor` views for editing text
- File input/output (I/O) in the mobile environment
- Asynchronous operations
- Property change notifications
- Data binding

The next chapter gets into multi-page architectures, page navigation, the powerful `ListView` for displaying collections of items, and dealing with application lifecycle issues.

Version 1. The Entry and Editor views

The **NoteTaker** app definitely requires some text input. Some phones have physical keyboards, but the vast majority are limited to virtual onscreen keyboards. Of course, these are somewhat different for each platform and often also vary by the type of text input.

Xamarin.Forms defines two views for obtaining text input:

- `Entry` for a single line of text
- `Editor` for multiple lines of text

Both `Entry` and `Editor` derive from `InputView`, which derives from `View`.

All three platforms have various styles of virtual keyboard appropriate for different types of text input. For example, a keyboard for typing a URL should be different from a keyboard for entering a phone number. For this reason, `InputView` defines a property named `Keyboard` of type `Keyboard`, a class that defines seven static read-only properties of type `Keyboard` appropriate for different keyboard uses:

- `Default`
- `Text`
- `Chat`
- `Url`
- `Email`

- Numeric
- Telephone

For example, if you have an Entry view intended for entering a URL, specify:

```
entry.Keyboard = Keyboard.Url;
```

Of course, the actual keyboards are platform-specific, and not all three platforms have distinct keyboards for all seven static properties.

The Keyboard class an alternative way to specify a keyboard using a KeyboardFlags enumeration, which has the following flag members:

- CapitalizeSentence (equal to 1)
- Spellcheck (2)
- Suggestions (4)
- All (\xFFFFFFFF)

These flags make more sense with an Editor rather than an Entry because the user is likely to be typing real text consisting of actual words organized into sentences. Set the keyboard like this:

```
editor.Keyboard = Keyboard.Create(KeyboardFlags.All);
```

The phone's operating system displays the virtual keyboard when the Entry or Editor acquires *keyboard input focus*. Input focus means that keyboard input is directed towards that particular view. Only one view can have input focus at any time. Although any Xamarin.Forms view can get input focus, only Entry and Editor are equipped to process that input.

A view must have its IsEnabled property set to true (the default state) to acquire input focus. A user can give an enabled view input focus by tapping it. When the Entry or Editor acquires input focus, the operating system pops up the keyboard.

A user can remove keyboard focus from an Entry or Editor by tapping somewhere else on the screen. The keyboard is automatically dismissed. But this is not the only way to dismiss the keyboard. Often a specific keyboard key or button associated with the keyboard lets the user signal that text input is complete. The concept of signaling that text is complete is complicated somewhat by the different nature of the Entry and Editor views: An **Enter** or **Return** key often removes input focus from an Entry view, but in an Editor that same key instead simply marks the end of one paragraph and the beginning of another.

On the iPhone, a **Return** button dismisses the keyboard from an Entry and a special **Done** button dismisses the keyboard from an Editor. On the Android, a **Done** key dismisses the keyboard from the Entry but a button on the bottom of the screen is required to dismiss the keyboard from the Editor. On the Windows Phone, the **Enter** key dismisses the keyboard from the Entry but the hardware **Back** button dismisses the keyboard from the Editor.

The Entry and Editor define Focused and Unfocused events that signal gaining and losing input focus. The IsFocused property indicates if a particular view currently has input focus. A program can attempt to set focus to a particular view in code by calling the Focus method on the view. The method returns true if the focus change was successful. These focus-related members are defined by Visual-Element, the base class for all UI elements, but they are really only relevant for the Entry and Editor.

Both Entry and Editor define Text properties that expose the current text displayed by the view. A program can initialize this Text property and then allow the user to edit that text.

Both Entry and Editor define a Completed event that is fired when the user has signaled that editing is completed. This is an excellent opportunity for programs to access the Text property and save the results.

Both Entry and Editor also define a TextChanged event that is fired when the Text property changes. (Keep in mind that .NET string objects are immutable. A string object can't itself change after it's been created. Every time the contents of the Entry or Editor change, it's a whole new string object rather than a modified string object.) The TextChanged event can be a valuable means for the application to monitor the text input on a character-by-character basis and respond to changes before the user signals that the typing is complete.

You might assume that Entry is somewhat simpler than Editor because it handles only a single line of text. However, Entry has three additional properties that Editor doesn't have:

- TextColor — a Color value
- IsPassword — a Boolean that causes characters to be masked
- Placeholder — light colored text that appears in the Entry but disappears as soon as the user begins typing.

The **NoteTaker** program requires a page that contains an Entry for the user to type a title for the note, and an Editor for typing the note itself. For the first version of this program—called **NoteTaker1**—let's not do much more beyond getting these two views on the page with a couple of Label views for identification.

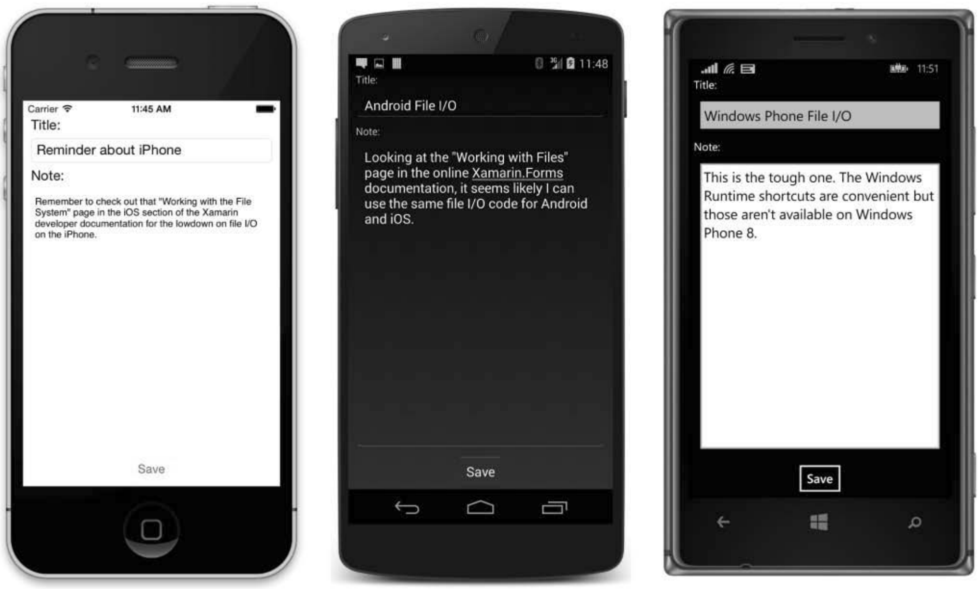
The Editor allows an indefinite amount of text to be entered and internally implements scrolling. As you discovered with the ScrollView in the previous chapter, sharing a scrollable view with other views on the page requires that the VerticalOptions property be set to LayoutOptions.-FillAndExpand.

Here's the NoteTaker1Page class:

```
class NoteTaker1Page : ContentPage
{
    public NoteTaker1Page()
    {
        this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);
        this.Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "Title:"
                },
                new Entry
                {
                    Placeholder = "Title (optional)"
                },
                new Label
                {
                    Text = "Note:"
                },
                new Editor
                {
                    Keyboard = Keyboard.Create(KeyboardFlags.All),
                    BackgroundColor = Device.OnPlatform(Color.Default,
                                                        Color.Default,
                                                        Color.White),
                    VerticalOptions = LayoutOptions.FillAndExpand
                },
                new Button
                {
                    Text = "Save",
                    HorizontalOptions = LayoutOptions.Center
                }
            }
        };
    }
}
```

The program explicitly sets the `BackgroundColor` property on the `Editor` to correct a little flaw in the Windows Phone implementation: Without this setting, the `Editor` background goes black when it loses input focus, and that causes the black text to be invisible.

Here's what it looks like with some text typed in:



Reminder about iPhone

Remember to check out that "Working with the File System" page in the iOS section of the Xamarin developer documentation for the lowdown on file I/O on the iPhone.

Android File I/O

Looking at the "Working with Files" page in the online Xamarin.Forms documentation, it seems likely I can use the same file I/O code for Android and iOS.

Windows Phone File I/O

This is the tough one. The Windows Runtime shortcuts are convenient but those aren't available on Windows Phone 8.

The `Button` at the bottom labeled **Save** currently does nothing. Such a button is undoubtedly intended to save the note to a file. To do that, however, we need to know how to save files, and later load them back in.

File input/output is not something included in the Xamarin.Forms API. Yet, some file I/O is required by many mobile apps, if only to save settings and interim data. For this reason, file I/O is probably the most compelling reason to skirt around the Xamarin.Forms API and implement some vital features of your app by using the programming interfaces of the individual platforms.

Version 2. File input/output

Traditionally, file input/output is one of the most basic programming tasks, but file I/O on mobile devices is a little different than on the desktop. On the desktop, users and applications generally have a whole disk available organized in a directory structure. On mobile devices, several standard folders exist—for pictures or music, for example—but application-specific data is generally restricted to private storage areas.

Programmers familiar with .NET know that the `System.IO` namespace contains the bulk of standard file I/O support. Perhaps the most important class in this namespace is the static `File` class, which not only provides a bunch of methods to create new files and open existing files but also includes several methods capable of performing an entire file read or write operation in a single method call.

For example, the `File.WriteAllText` method has two arguments of type `string`—a filename and the file contents. The method creates the file (replacing an existing file with the same name if necessary), writes the contents to the file, and closes it. The `File.ReadAllText` method is similar but returns the contents of the file in one big `string` object. These methods seem ideal for the job of saving and retrieving notes.

The Xamarin.iOS and Xamarin.Android libraries include a version of .NET that has been expressly tailored by Xamarin for these two mobile platforms. The methods in the `File` class in the `System.IO` namespace map to appropriate file I/O functions in the iOS and Android platforms. This means that you can use methods in the `File` class—including `File.WriteAllText` and `File.ReadAll-Text`—in your iPhone and Android applications.

Let's experiment a bit:

Go into Visual Studio or Xamarin Studio, and load any Xamarin.Forms solution created so far, such as **NoteTaker1**. Bring up one of the code files in the iOS or Android project. In a constructor or method, type the `System.IO` namespace name and then a period. You'll get a list of all the available types in the namespace. If you then type `File` and a period, you'll get all the static methods in the `File` class, including `WriteAllText` and `ReadAllText`.

In a Windows Phone project, however, you're working with a version of .NET created by Microsoft and stripped down somewhat for the Windows Phone platform. If you type `System.IO.File` and a period, you'll see a rather diminished `File` class that does *not* include `WriteAllText` and `ReadAll-Text`, although it does include methods to create and open text files.

Now go into any code file in a Xamarin.Forms Portable Class Library project, and type `System.IO` and a period. Now you won't even see the `File` class! It does not exist in the PCL. Why is that? PCLs are configured to support multiple target platforms. The APIs implemented within the PCL are necessarily an intersection of the APIs in these target platforms.

A PCL appropriate for Xamarin.Forms includes the following platforms:

- .NET Framework 4.5
- Windows 8
- Windows Phone Silverlight 8
- Xamarin.Android
- Xamarin.iOS

Notice the inclusion of Windows 8, which incorporates an API called the Windows Runtime (or WinRT). Microsoft completely revamped file I/O for WinRT and created a whole new file I/O API. The `System.IO.File` class does not exist in the PCL because it is not part of WinRT.

Although the `File` class does not exist in a Portable Class Library project, you might wonder what kind of `File` class you can use in a Shared Asset Project. Well, it varies by what platform is being compiled. You can use `File.WriteAllText` and `File.ReadAllText` in your iOS and Android projects but not in your Windows Phone projects. Your Windows Phone projects need something else.

Skip past the scary stuff?

Already you might suspect that this subject of file I/O is going to get hairy, and you are correct. But consider what we're trying to do here: We're trying to target three different mobile platforms with a common code base. That's not easy, and we're bound to encounter some rough terrain along the way.

So the question has to be: Metaphorically speaking, do you enjoy hiking through treacherous terrain to climb to the top of a mountain and get a gorgeous view? Or would you prefer that somebody else takes a photo from the top of the mountain and sends it to you in an email?

If you're in the latter category, you might prefer to skip over much of this chapter and jump straight to the comparatively simple and elegant solution to the problem of Xamarin.Forms file I/O presented at the chapter's end and continuing into the next chapter. Until that point, some of the transitional code you'll encounter will be both scary and ugly.

If you choose to take this long path up the mountain, you'll understand why the trip is necessary, and you'll understand the rationale behind the platform differences—even in seemingly routine jobs like file I/O.

In the previous chapter you saw how the Xamarin.Forms `Device` class can be a valuable tool for dealing with platform differences. But the code that's referenced by the `Device` class must be compilable in all three platforms. This is not the case for file I/O because the different platforms have access to different APIs. This means that the platform differences can't be managed using the `Device` class and must be handled in other ways. Moreover, the solutions are different for Shared Asset Projects and Portable Class Libraries projects.

For this reason, for the next two versions of **NoteTaker**, there will be separate solutions for SAP and PCL. The two different solutions for version 2 are named **NoteTaker2Sap** and **NoteTaker2Pcl**.

Preprocessing in the SAP

Dealing with platform differences is a Shared Asset Project is a little more straightforward than a PCL and involves more traditional programming tools, so let's begin with that.

In code files in a Shared Asset Project, you can use the C# preprocessor directives `#if`, `#elif`, `#else`, and `#endif` with conditional compilation symbols defined for the three platforms. These symbols are __IOS__ for iOS and WINDOWS_PHONE for Windows Phone; there are no conditional compilation symbols for Android, but Android can be identified as not being iOS or Windows Phone.

The **NoteTaker2Sap** project includes a class named `FileHelper` in a file named FileHelper.cs. You can add such a file to the project the same way you add a new file for the class that derives from `ContentPage`.

The FileHelper.cs file uses C# preprocessor directives to divide the code into two sections. The first section is for iOS and Android and is compiled if the WINDOWS_PHONE identifier is not defined. The second section is for Windows Phone.

Both sections contain three public static methods named Exists, WriteAllText, and ReadAllText. In the first section, the iOS and Android versions of these functions use standard static File methods but with a folder name obtained from the Environment.GetFolderPath method with an argument of Environment.SpecialFolder.MyDocuments:

```
namespace NoteTaker2Sap
{
    static class FileHelper
    {

#if !WINDOWS_PHONE // iOS and Android

        public static bool Exists(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.Exists(filepath);
        }

        public static void WriteAllText(string filename, string text)
        {
            string filepath = GetFilePath(filename);
            File.WriteAllText(filepath, text);
        }

        public static string ReadAllText(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.ReadAllText(filepath);
        }

        static string GetFilePath(string filename)
        {
            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
            return Path.Combine(docsPath, filename);
        }

#else // Windows Phone

        public static bool Exists(string filename)
        {
            return File.Exists(filename);
        }

        public static void WriteAllText(string filename, string text)
        {
            StreamWriter writer = File.CreateText(filename);
            writer.Write(text);
            writer.Close();
        }

        public static string ReadAllText (string filename)
        {
            StreamReader reader = File.OpenText(filename);
            string text = reader.ReadToEnd();
            reader.Close();
            return text;
        }

#endif

    }
}
```

When the Shared Asset Project is compiled for Windows Phone, the `File.WriteAllText` and `File.ReadAllText` methods do not exist so those can't appear in the Windows Phone section. However, static `CreateText` and `OpenText` methods are available, and these are used to obtain `StreamWriter` and `StreamReader` objects. This Windows Phone code works in the sense that it doesn't raise an exception, but you'll see shortly that it really doesn't do what you want. Something else is required.

Besides the `FileHelper` class to handle low-level file I/O, the **NoteTaker2Sap** project includes another new class named `Note`. This class encapsulates the two `string` objects associated with a note in simple read/write properties named `Title` and `Text`. This class also includes methods named `Save` and `Load` that call the appropriate methods in `FileHelper`:

```
namespace NoteTaker2Sap
{
    class Note
    {
        public string Title { set; get; }

        public string Text { set; get; }

        public void Save(string filename)
        {
            string text = this.Title + "\n" + this.Text;
            FileHelper.WriteAllText(filename, text);
        }

        public void Load(string filename)
        {
            string text = FileHelper.ReadAllText(filename);

            // Break string into Title and Text.
            int index = text.IndexOf('\n');
            this.Title = text.Substring(0, index);
            this.Text = text.Substring(index + 1);
        }
    }
}
```

Notice that the `Save` method simply joins the two `string` objects into one with a line feed character, and the `Load` method takes them apart.

Finally, the `NoteTaker2SapPage` class has a very similar page layout as the first program but contains two buttons labeled **Save** and **Load** that use the `Note` class for these operations. A filename of “test.note” is used throughout:

```
class NoteTaker2SapPage : ContentPage
{
    static readonly string FILENAME = "test.note";

    Entry entry;
    Editor editor;
    Button loadButton;

    public NoteTaker2SapPage()
    {
        // Create Entry and Editor views.
        entry = new Entry
        {
            Placeholder = "Title (optional)"
        };

        editor = new Editor
        {
            Keyboard = Keyboard.Create(KeyboardFlags.All),
            BackgroundColor = Device.OnPlatform(Color.Default,
                                                Color.Default,
                                                Color.White),
            VerticalOptions = LayoutOptions.FillAndExpand
        };

        // Create Save and Load buttons.
        Button saveButton = new Button
        {
            Text = "Save",
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };
        saveButton.Clicked += OnSaveButtonClicked;

        loadButton = new Button
        {
            Text = "Load",
            IsEnabled = FileHelper.Exists(FILENAME),
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };
        loadButton.Clicked += OnLoadButtonClicked;

        // Assemble page.
        this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);

        this.Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "Title:"
                },
                entry,
                new Label
                {
                    Text = "Note:"
                },
                editor,
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    Children =
                    {
                        saveButton,
                        loadButton
                    }
                }
            }
        };
    }

    void OnSaveButtonClicked(object sender, EventArgs args)
    {
        Note note = new Note
        {
            Title = entry.Text,
            Text = editor.Text
        };
        note.Save(FILENAME);
        loadButton.IsEnabled = true;
    }

    void OnLoadButtonClicked(object sender, EventArgs args)
    {
        Note note = new Note();
        note.Load(FILENAME);
        entry.Text = note.Title;
        editor.Text = note.Text;
    }
}
```

Notice that the **Load** button initialization calls `FileHelper.Exists` to determine if the file exists and disables the button if it does not. The button is then enabled the first time the file is saved.

You'll want to convince yourself that this works—that the information is saved to the device (or phone simulator). Type something into the `Entry` and `Editor`, and then press **Save** to save that information. Clear out the `Entry` and `Editor` (or type in new text), and press **Load** to restore the information that was saved.

Here's what's really important: If you terminate the program or shut down the phone and then rerun the program, the saved file still exists.

Well, in two out of three cases, the saved file still exists. It works with iOS and Android, but not Windows Phone. Although the Windows Phone **Save** and **Load** buttons seem to work while the program is running, the file is not persisted when the application is exited. Getting Windows Phone to work right will require a different set of file I/O classes.

Meanwhile, let's try to get this simple (two-thirds functional) version to work with a Portable Class Library solution.

Dependency service in the PCL

As you've seen, the `System.IO.File` class does not exist in the version of .NET available to a Xamarin.Forms Portable Class Library. This means that if you've created a PCL-based Xamarin.Forms solution, the file I/O code cannot be in the PCL. The file I/O code must be implemented in the individual platform projects where it can use the version of .NET specifically for that platform.

Yet, the PCL must somehow make calls to these file I/O functions. Normally that wouldn't work: Application projects make calls to libraries all the time, but libraries generally can't make calls to applications except with events or callback functions.

It is the main purpose of the Xamarin.Forms `DependencyService` class to get around this restriction. Although this class is implemented in the Xamarin.Forms.Core library assembly and used in a PCL, it uses .NET reflection to search through all the other assemblies available in the application, including the particular platform-specific application assembly itself. (It is also possible for the platform projects to use dependency injection techniques to configure the PCL to make calls into the platform projects.)

To use `DependencyService`, the first requirement is that the PCL must contain an interface definition that includes the names and signatures of the platform-specific methods you need. Here is that file in the **NoteTaker2Pcl** project:

```
namespace NoteTaker2Pcl
{
    public interface IFileHelper
    {
        bool Exists(string filename);

        void WriteAllText(string filename, string text);

        string ReadAllText(string filename);
    }
}
```

This interface must be public to the PCL because it must be visible to the individual platform projects.

Next, in all three application projects, you create code files with classes that implement this interface. Here's the one in the iOS project. (You can tell that this file is in the iOS project by the namespace):

```
using System;
using System.IO;
using Xamarin.Forms;
```

```
[assembly: Dependency(typeof(NoteTaker2Pcl.iOS.FileHelper))]

namespace NoteTaker2Pcl.iOS
{
    class FileHelper : IFileHelper
    {
        public bool Exists(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.Exists(filepath);
        }

        public void WriteAllText(string filename, string text)
        {
            string filepath = GetFilePath(filename);
            File.WriteAllText(filepath, text);
        }

        public string ReadAllText(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.ReadAllText(filepath);
        }

        string GetFilePath(string filename)
        {
            string docsPath = Environment.GetFolderPath(
                        Environment.SpecialFolder.MyDocuments);
            return Path.Combine(docsPath, filename);
        }
    }
}
```

The actual implementation of these methods involves the same code that you've already seen except with instance methods rather than static methods. But take note of two necessary characteristics of this file:

- The class implements the `IFileHelper` interface defined in the PCL. Because it implements that interface, the three methods defined in the interface must be defined as public in this class.
- A special assembly-level attribute named `Dependency` is defined prior to the `namespace` definition.

`Dependency` is a special Xamarin.Forms attribute defined by the `DependencyAttribute` class specifically for use with the `DependencyService` class. The `Dependency` attribute simply specifies the type of the class but it assists the `DependencyService` class in locating the implementation of the interface in the application projects.

A similar file is in the Android project:

```
using System;
using System.IO;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker2Pcl.Droid.FileHelper))]

namespace NoteTaker2Pcl.Droid
{
    class FileHelper : IFileHelper
    {
        public bool Exists(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.Exists(filepath);
        }

        public void WriteAllText(string filename, string text)
        {
            string filepath = GetFilePath(filename);
            File.WriteAllText(filepath, text);
        }

        public string ReadAllText(string filename)
        {
            string filepath = GetFilePath(filename);
            return File.ReadAllText(filepath);
        }

        string GetFilePath(string filename)
        {
            string docsPath = Environment.GetFolderPath(
                        Environment.SpecialFolder.MyDocuments);
            return Path.Combine(docsPath, filename);
        }
    }
}
```

And in the Windows Phone project:

```
using System;
using System.IO;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker2Pcl.WinPhone.FileHelper))]

namespace NoteTaker2Pcl.WinPhone
{
    class FileHelper : IFileHelper
    {
        public bool Exists(string filename)
        {
            return File.Exists(filename);
        }

        public void WriteAllText(string filename, string text)
        {
            StreamWriter writer = File.CreateText(filename);
            writer.Write(text);
            writer.Close();
        }

        public string ReadAllText(string filename)
        {
            StreamReader reader = File.OpenText(filename);
            string text = reader.ReadToEnd();
            reader.Close();
            return text;
        }
    }
}
```

Now the hard work is done. You'll recall that the `Note` class in **NoteTaker2Sap** made calls to `FileHelper.WriteAllText` and `FileHelper.ReadAllText`. The `Note` class in **NoteTaker2Pcl** is very similar but instead references the two methods through the static `DependencyService.Get` method. This is a generic method that requires the interface as a generic argument but then is capable of calling any method in that interface:

```
namespace NoteTaker2Pcl
{
    class Note
    {
        public string Title { set; get; }

        public string Text { set; get; }

        public void Save(string filename)
        {
            string text = this.Title + "\n" + this.Text;
            DependencyService.Get<IFileHelper>().WriteAllText(filename, text);
        }

        public void Load(string filename)
        {
            string text = DependencyService.Get<IFileHelper>().ReadAllText(filename);

            // Break string into Title and Text.
            int index = text.IndexOf('\n');
            this.Title = text.Substring(0, index);
            this.Text = text.Substring(index + 1);
        }
    }
}
```

Internally, the `DependencyService` class searches for the interface implementation in the particular platform project and makes a call to the specified method.

The `NoteTaker2PclPage` class is nearly the same as `NoteTaker2SapPage` except it also uses `DependencyService.Get` to call the `Exists` method during the initialization of the **Load** button.

```
    loadButton = new Button
    {
        Text = "Load",
        IsEnabled = DependencyService.Get<IFileHelper>().Exists(FILENAME),
        HorizontalOptions = LayoutOptions.CenterAndExpand
    };
```

Of course, the **NoteTaker2Pcl** version has the same deficiency as **NoteTaker2Sap** in that it doesn't persist the data for Windows Phone.

Version 3. Going async

The data isn't persisted for Windows Phone because the file I/O code is not correct. A Xamarin.Forms application targets Windows Phone 8, which implements a subset of the same WinRT file I/O available to Windows 8 applications, largely found in the new `Windows.Storage` and `Windows.Storage.-Streams` namespaces.

Windows Phone 8 continues to support some older file I/O functions that Windows Phone 7 inherited from Silverlight, but these are not recommended for new Windows Phone 8 applications. Windows Phone 8 applications should instead use the WinRT file I/O API, and the programs in this book follow that recommendation.

Part of the impetus behind this new array of file I/O classes in Windows 8 and Windows Phone 8 is a recognition of a transition away from the relatively unconstrained file access of desktop applications towards a more sandboxed environment. To store data that is private to an application, a Windows Phone program first gets a special `StorageFolder` object:

```
StorageFolder localFolder = ApplicationData.Current.LocalFolder;
```

`ApplicationData` has a static property named `Current` that returns the `ApplicationData` object for the application. `LocalFolder` is an instance property of `ApplicationData`.

`StorageFolder` defines methods named `CreateFileAsync` to create a new file and `GetFileAsync` to open an existing file. These two methods return objects of type `StorageFile`. At this point, a program can open the file for writing or reading with `OpenAsync` or `OpenReadAsync`. These methods return an `IRandomAccessStream` object. From this, `DataWriter` or `DataReader` objects are created to perform write or read operations.

This sounds a bit lengthy, and it is. A rather simpler approach for text files involves the static methods `FileIO.ReadTextAsync` and `FileIO.WriteTextAsync`. The first argument to these methods is a `StorageFile` object, and the methods incorporate all the operations to open the file, write to it or read from it, and close the file. Although these methods are available in Windows Phone 8.1, they are not in Windows Phone 8, the version of Windows Phone supported by Xamarin.Forms.

At any rate, by this time you've undoubtedly noticed the frequent `Async` suffix on these method names. These are asynchronous methods. Internally, these methods spin off secondary threads of execution for doing the actual work and return quickly to the caller. The work takes place in the background, and when that work is finished—when the file has been created or opened, or written to or read from—the caller is notified through a call-back function and provided with the function's result.

Which might raise the question: *Why* are these methods asynchronous? Why are they more complex than the old .NET file I/O functions?

Graphical user interfaces have an intrinsic problem. Although an application can consist of multiple threads of execution, access to the user interface must usually be restricted to a single thread. The problem is not so much the graphical output, but the user input, which in various environments might include the keyboard, mouse, pen, or touch. In general, it can't be known where a user input event should be routed until all previous user input events have been processed. This means that user input events must be processed sequentially in a single thread of execution.

The impact of this restriction is profound: In the general case, *all* the application's user interface processing must be handled in a single thread, often called the *UI thread*. Even the seemingly innocent act of using a secondary thread of execution to set a property of a user-interface object such as `Entry` or `Editor` is forbidden.

At the same time, programmers are cautioned against doing any lengthy processing in this UI thread. If the UI thread is carrying out some lengthy processing, it can't respond to user input events and the entire user interface can seem to freeze up.

As we users have become more accustomed to graphical user interfaces over the decades, we've become increasingly intolerant of even the slightest lapse in responsiveness. Consequently, as application programmers, we are increasingly encouraged to avoid lengthy processing in the UI thread and to keep the application as responsive as possible.

This implies that lengthy processing jobs should be relegated to secondary threads of execution. These threads run "in the background," asynchronously with the UI thread.

The future of computing will undoubtedly involve a lot more asynchronous computing and parallel processing, particularly with the increasing use of multicore processor chips. Developers will need good language tools to work with asynchronous operations, and fortunately C# has been in the forefront in this regard.

When the WinRT APIs used for Windows 8 Store apps were being developed, the Microsoft developers took a good hard look at timing and decided that any function call that could require more than 50 milliseconds to execute should be made asynchronous so that it would not interfere with the responsiveness of the user interface.

APIs that require more than 50 milliseconds obviously include file I/O functions, which often need to access potentially slow pieces of hardware, like disk drives or a network. Any WinRT file I/O function that could possibly hit a physical storage device was made asynchronous and given an `Async` method suffix.

The CreateFileAsync method defined by the StorageFolder class does not directly return a StorageFile object. Instead, it returns an IAsyncOperation<StorageFile> object:

```
IAsyncOperation<StorageFile> createOp = storageFolder.CreateFileAsync("filename");
```

The IAsyncOperation interface, its base interface IAsyncInfo, and related interfaces such as IAsyncAction, are all defined in the Windows.Foundation namespace, indicating how fundamental they are to the entire operating system. A return value such as IAsyncOperation is sometimes referred to as a "promise." The StorageFile object is not available just yet, but it will be in the future if nothing goes awry.

To begin the actual asynchronous operation, you must assign a handler to the Completed property of the IAsyncOperation object:

```
createOp.Completed = OnCreateFileCompleted;
```

Completed is a property rather than an event but it functions much like an event. The big difference is that Completed can't have multiple handlers. Assigning a callback method (named OnCreateFile-Completed in this example) actually initiates the background process.

The code that sets the Completed property to a handler executes very quickly, and the program can then continue normally. Simultaneously, the file is being created in a secondary thread. When the file is created, that background code calls the callback method assigned to the Completed handler in your code. That callback method might look like this:

```
void OnCreateFileCompleted(IAsyncOperation<StorageFile> createOp, AsyncStatus asyncStatus)
{
    if (asyncStatus == AsyncStatus.Completed)
    {
        StorageFile storageFile = createOp.GetResults();
        // continue with next step ...
    }
    else
    {
        // deal with cancellation or error
    }
}
```

The second argument indicates the status, and at this point it's either Completed, Canceled, or Error. Members of the first argument can provide more detail about any error that might have occurred. If all is well, calling GetResults on the first argument obtains the StorageFile object.

At this point, the next step would be to open that file for writing. A call to OpenAsync returns an object of type IAsyncOperation<IRandomAccessStream>, and that involves another callback method:

```
void OnCreateFileCompleted(IAsyncOperation<StorageFile> createOp, AsyncStatus asyncStatus)
{
    if (asyncStatus == AsyncStatus.Completed)
    {
        StorageFile storageFile = createOp.GetResults();
        IAsyncOperation<IRandomAccessStream> openOp =
                    storageFile.OpenAsync(FileAccessMode.ReadWrite);
        openOp.Completed = OnOpenFileCompleted;
    }
    else
    {
        // deal with cancellation or error
    }
}

void OnOpenFileCompleted(IAsyncOperation<IRandomAccessStream> openOp, AsyncStatus asyncStatus)
{
    // ...
}
```

One way to simplify this code is to use anonymous lambda functions for the callbacks. This avoids a proliferation of individual methods and allows more free-form access to local variables. But for a sequence of asynchronous method calls, it tends to produce a nested structure of asynchronous callbacks, as you'll see.

Asynchronous lambdas in the SAP

In the **NoteTaker3Sap** project, the file I/O code has been moved to the `Note` class and performed separately for Windows Phone using lambda functions for callbacks. To keep the code simple (at least comparatively so), there is no error handling:

```
using System;

#if !WINDOWS_PHONE
using System.IO;
#else
using Windows.Foundation;
using Windows.Storage;
using Windows.Storage.Streams;
#endif

namespace NoteTaker3Sap
{
    class Note
    {
        public string Title { set; get; }

        public string Text { set; get; }

        public void Save(string filename)
        {
            string text = this.Title + "\n" + this.Text;

#if !WINDOWS_PHONE // iOS and Android

            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
            string filepath = Path.Combine(docsPath, filename);
            File.WriteAllText(filepath, text);

#else // Windows Phone

            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IAsyncOperation<StorageFile> createOp =
                            localFolder.CreateFileAsync(filename,
                                CreationCollisionOption.ReplaceExisting);

            createOp.Completed = (asyncInfo1, asyncStatus1) =>
            {
                IStorageFile storageFile = asyncInfo1.GetResults();
                IAsyncOperation<IRandomAccessStream> openOp =
                                       storageFile.OpenAsync(FileAccessMode.ReadWrite);
                openOp.Completed = (asyncInfo2, asyncStatus2) =>
                {
                    IRandomAccessStream stream = asyncInfo2.GetResults();
                    DataWriter dataWriter = new DataWriter(stream);
                    dataWriter.WriteString(text);
                    DataWriterStoreOperation storeOp = dataWriter.StoreAsync();
                    storeOp.Completed = (asyncInfo3, asyncStatus3) =>
                    {
                        dataWriter.Dispose();
                    };
                };
            };

#endif
        }

        public void Load(string filename)
        {

#if !WINDOWS_PHONE // iOS and Android

            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
            string filepath = Path.Combine(docsPath, filename);
            string text = File.ReadAllText(filepath);

            // Break string into Title and Text.
            int index = text.IndexOf('\n');
            this.Title = text.Substring(0, index);
            this.Text = text.Substring(index + 1);

#else // Windows Phone

            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IAsyncOperation<StorageFile> createOp = localFolder.GetFileAsync(filename);
            createOp.Completed = (asyncInfo1, asyncStatus1) =>
            {
                IStorageFile storageFile = asyncInfo1.GetResults();
                IAsyncOperation<IRandomAccessStreamWithContentType> openOp =
                                        storageFile.OpenReadAsync();
                openOp.Completed = (asyncInfo2, asyncStatus2) =>
                {
                    IRandomAccessStream stream = asyncInfo2.GetResults();
                    DataReader dataReader = new DataReader(stream);
                    uint length = (uint)stream.Size;
                    DataReaderLoadOperation loadOp = dataReader.LoadAsync(length);
                    loadOp.Completed = (asyncInfo3, asyncStatus3) =>
                    {
                        string text = dataReader.ReadString(length);
                        dataReader.Dispose();

                        // Break string into Title and Text.
                        int index = text.IndexOf('\n');
                        this.Title = text.Substring(0, index);
                        this.Text = text.Substring(index + 1);
                    };
                };
            };

#endif

        }
    }
}
```

Notice the `Dispose` calls on the `DataWriter` and `DataReader` methods. It might be tempting to remove these calls under the assumption that the objects are disposed automatically when the objects go out of scope, but this is not the case. If `Dispose` is not called, the files remain open,

But the big question is: Why has all this code been moved to the `Note` class? Why isn't it isolated in a separate `FileHelper` class as in the previous version of the program?

The problem is that a method that requires asynchronous callbacks to obtain an object can't directly return that object to the caller. Look at the `Load` method here. When that `Load` method is called in a Windows Phone program, the `localFolder` variable is set, and the `createOp` object is set, but as soon as the `Completed` property is set to the asynchronous callback method, the `Load` method returns to the caller. But the method doesn't yet have anything to return! The `GetFileAsync` operation is proceeding in the background in a secondary thread. Only later does the `Completed` callback method execute for the next step of the job. When the contents of the file are finally read within these nested callbacks, the contents must be stored somewhere. Fortunately, the code is in the `Note` class, so the results can be stored in the `Title` and `Text` properties.

In the previous version of this program, the `Exists` method returned a Boolean to indicate the existence of a file. That code needs to be moved to the `NoteTaker3SapPage` class where it has access to the `Button` whose `IsEnabled` property must be set:

```
using System;

#if !WINDOWS_PHONE
using System.IO;
#else
```

```
using Windows.Foundation;
using Windows.Storage;
using Windows.Storage.Streams;
#endif

using Xamarin.Forms;

namespace NoteTaker3Sap
{
    class NoteTaker3SapPage : ContentPage
    {
        static readonly string FILENAME = "test.note";

        Note note = new Note();
        Entry entry;
        Editor editor;
        Button loadButton;

        public NoteTaker3SapPage()
        {
            // Create Entry and Editor views.
            entry = new Entry
            {
                Placeholder = "Title (optional)"
            };

            editor = new Editor
            {
                Keyboard = Keyboard.Create(KeyboardFlags.All),
                BackgroundColor = Device.OnPlatform(Color.Default,
                                                    Color.Default,
                                                    Color.White),
                VerticalOptions = LayoutOptions.FillAndExpand
            };

            // Create Save and Load buttons.
            Button saveButton = new Button
            {
                Text = "Save",
                HorizontalOptions = LayoutOptions.CenterAndExpand
            };
            saveButton.Clicked += OnSaveButtonClicked;

            loadButton = new Button
            {
                Text = "Load",
                IsEnabled = false,
                HorizontalOptions = LayoutOptions.CenterAndExpand
            };
            loadButton.Clicked += OnLoadButtonClicked;

            // Check if the file is available.

#if !WINDOWS_PHONE // iOS and Android
```

```
            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.Personal);
            string filepath = Path.Combine(docsPath, FILENAME);
            loadButton.IsEnabled = File.Exists(filepath);

#else // Windows Phone

            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IAsyncOperation<StorageFile> createOp = localFolder.GetFileAsync(FILENAME);
            createOp.Completed = (asyncInfo, asyncStatus) =>
            {
                loadButton.IsEnabled = asyncStatus != AsyncStatus.Error;
            };

#endif

            // Assemble page.
            this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);

            this.Content = new StackLayout
            {
                Children =
                {
                    new Label
                    {
                        Text = "Title:"
                    },
                    entry,
                    new Label
                    {
                        Text = "Note:"
                    },
                    editor,
                    new StackLayout
                    {
                        Orientation = StackOrientation.Horizontal,
                        Children =
                        {
                            saveButton,
                            loadButton
                        }
                    }
                }
            };
        }

        void OnSaveButtonClicked(object sender, EventArgs args)
        {
            note.Title = entry.Text;
            note.Text = editor.Text;
            note.Save(FILENAME);
            this.loadButton.IsEnabled = true;
        }
        void OnLoadButtonClicked(object sender, EventArgs args)
        {
            note.Load(FILENAME);
            entry.Text = note.Title;
            editor.Text = note.Text;
        }
    }
}
```

To set that IsEnabled property of the **Load** button in the Windows Phone version, the strategy is to attempt to call GetFileAsync. If that call reports an error in the asynchronous callback, the file does not exist. (The StorageFile class defines an IsAvailable property, but it isn't supported on Windows Phone.)

Notice that this version of the page class contains a single Note object instantiated as a field and accessed by both Button event handlers. This makes more sense than creating a new Note object in each call to the Clicked handlers. In the final version of the program, when a page like this is used for creating a new note or editing an existing note, the page and the Note object will exist as a tightly-linked pair—one class for the user interface, and another class for the underlying data exposed by the user interface.

If you try out this program on Windows Phone, you'll still discover a problem:

Type some text in the Entry and Editor. Press the **Save** button. Now erase that text or change it. Press the **Load** button. The saved text returns. Great!

Now end the program and start it up again. The Entry and Editor fields are blank but the **Load** button is enabled. Excellent! The file still exists. Press the **Load** button. Nothing. That's odd: If the Button is enabled, the file should exist, so where is it? Now press the **Load** button a second time. Ahh, there it is!

Can you figure out what's happening?

When you run the program and press the **Load** button to load a previously created file, the **Load** method in Note is called. But that method returns after calling GetFileAsync. The file hasn't been read yet, the Title and Text properties of Note haven't yet been set, but the OnLoadButton-Clicked method blithely sets the contents of those Title and Text properties to the Entry and Editor. The callbacks in the Load method in Note continue to execute until the file is read and the Title and Text properties are eventually set, so pressing the **Load** button a second retrieves them.

This problem shows up only when the program starts up, because thereafter the values in the Note object are always the last values saved to the file.

Method callbacks in the PCL

Can these asynchronous method calls be incorporated in a PCL project that uses the Dependency-Service to access platform-specific versions of the file I/O logic? Certainly not in the same form as in the SAP version. The Exists and ReadAllText methods must return values—a bool and a string, respectively—and we've already seen that the asynchronous function calls in a method can't return values.

But these methods *can* return values if they return those values in their own callback functions!

Here's the new `IFileHelper` interface in the **NoteTaker3Pcl** project:

```
using System;

namespace NoteTaker3Pcl
{
    public interface IFileHelper
    {
        void Exists(string filename, Action<bool> completed);

        void WriteAllText(string filename, string text, Action completed);

        void ReadAllText(string filename, Action<string> completed);
    }
}
```

All three methods now have a return type of `void`, but they all have a last argument that is a delegate for a method with (respectively) one Boolean argument, no arguments, and one `string` argument.

The iOS implementation of this interface is very similar to the previous version except that the `completed` method is called to indicate completion and to return any value:

```
using System;
using System.IO;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker3Pcl.iOS.FileHelper))]

namespace NoteTaker3Pcl.iOS
{
    class FileHelper : IFileHelper
    {
        public void Exists(string filename, Action<bool> completed)
        {
            bool exists = File.Exists(GetFilePath(filename));
            completed(exists);
        }

        public void WriteAllText(string filename, string text,
                                 Action completed)
        {
            File.WriteAllText(GetFilePath(filename), text);
            completed();
        }

        public void ReadAllText(string filename, Action<string> completed)
        {
            string text = File.ReadAllText(GetFilePath(filename));
            completed(text);
        }

        string GetFilePath(string filename)
        {
            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
            return Path.Combine(docsPath, filename);
        }
    }
}
```

The Android version is very similar. The Windows Phone version has methods that call the `completed` function in the innermost nested asynchronous callback:

```
using System;
using Windows.Foundation;
using Windows.Storage;
using Windows.Storage.Streams;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker3Pcl.WinPhone.FileHelper))]

namespace NoteTaker3Pcl.WinPhone
{
    class FileHelper : IFileHelper
    {
        public void Exists(string filename, Action<bool> completed)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IAsyncOperation<StorageFile> createOp = localFolder.GetFileAsync(filename);
            createOp.Completed = (asyncInfo, asyncStatus) =>
            {
                completed(asyncStatus != AsyncStatus.Error);
            };
        }

        public void WriteAllText(string filename, string text, Action completed)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IAsyncOperation<StorageFile> createOp =
                            localFolder.CreateFileAsync(filename,
                                CreationCollisionOption.ReplaceExisting);
            createOp.Completed = (asyncInfo1, asyncStatus1) =>
            {
                IStorageFile storageFile = asyncInfo1.GetResults();
                IAsyncOperation<IRandomAccessStream> openOp =
                        storageFile.OpenAsync(FileAccessMode.ReadWrite);
                openOp.Completed = (asyncInfo2, asyncStatus2) =>
                {
                    IRandomAccessStream stream = asyncInfo2.GetResults();
                    DataWriter dataWriter = new DataWriter(stream);
                    dataWriter.WriteString(text);
                    DataWriterStoreOperation storeOp = dataWriter.StoreAsync();
                    storeOp.Completed = (asyncInfo3, asyncStatus3) =>
                    {
```

```
                    dataWriter.Dispose();
                    completed();
                };
            };
        };
    }

    public void ReadAllText(string filename, Action<string> completed)
    {
        StorageFolder localFolder = ApplicationData.Current.LocalFolder;
        IAsyncOperation<StorageFile> createOp = localFolder.GetFileAsync(filename);
        createOp.Completed = (asyncInfo1, asyncStatus1) =>
        {
            IStorageFile storageFile = asyncInfo1.GetResults();
            IAsyncOperation<IRandomAccessStreamWithContentType> openOp =
                                        storageFile.OpenReadAsync();
            openOp.Completed = (asyncInfo2, asyncStatus2) =>
            {
                IRandomAccessStream stream = asyncInfo2.GetResults();
                DataReader dataReader = new DataReader(stream);
                uint length = (uint)stream.Size;
                DataReaderLoadOperation loadOp = dataReader.LoadAsync(length);
                loadOp.Completed = (asyncInfo3, asyncStatus3) =>
                {
                    string text = dataReader.ReadString(length);
                    dataReader.Dispose();
                    completed(text);
                };
            };
        };
    }
}
}
```

To hide away the calls to the `DependencyService.Get` method, another file has been added to the **NoteTaker3Pcl** project. This class lets other code in the program use normal-looking static `FileHelper` methods:

```
namespace NoteTaker3Pcl
{
    static class FileHelper
    {
        static IFileHelper fileHelper = DependencyService.Get<IFileHelper>();

        public static void Exists(string filename, Action<bool> completed)
        {
            fileHelper.Exists(filename, completed);
        }

        public static void WriteAllText(string filename, string text, Action completed)
        {
            fileHelper.WriteAllText(filename, text, completed);
        }
        public static void ReadAllText(string filename, Action<string> completed)
        {
            fileHelper.ReadAllText(filename, completed);
        }
    }
}
```

Notice that the class also saves the `DependencyService` object associated with the `IFileHelper` interface in a static field to make the actual calls more efficient. Although this class seems to implement the `IFileHelper` interface, it actually does not implement that interface because the class and methods are all static.

The `Note` class is now almost as simple as the original version. The only real difference is a `Load` method that sets the `Title` and `Text` fields in a lambda function passed to the `FileHelper.ReadAllText` method:

```
namespace NoteTaker3Pcl
{
    class Note
    {
        public string Title { set; get; }

        public string Text { set; get; }

        public void Save(string filename)
        {
            string text = this.Title + "\n" + this.Text;
            FileHelper.WriteAllText(filename, text, () => { });
        }

        public void Load(string filename)
        {
            FileHelper.ReadAllText(filename, (string text) =>
                {
                    // Break string into Title and Text.
                    int index = text.IndexOf('\n');
                    this.Title = text.Substring(0, index);
                    this.Text = text.Substring(index + 1);
                });
        }
    }
}
```

The only difference in the page file is the code to determine if the **Load** button should be disabled. The `IsEnabled` setting occurs in a lambda function passed to the `FileHelper.Exists` method:

```
loadButton = new Button
{
    Text = "Load",
    IsEnabled = false,
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
loadButton.Clicked += OnLoadButtonClicked;
```

```
// Check if the file is available.
FileHelper.Exists(FILENAME, (exists) =>
    {
        loadButton.IsEnabled = exists;
    });
```

Does this fix the problem with the first press of the **Load** button on the Windows Phone? No, it does not. The `OnLoadButtonClicked` method is still setting the `Entry` and `Editor` to text from `Note` class properties before that text has been loaded from the file. To work properly, that code would need to know when the `Note` properties were set before transferring them to the `Editor` and `Entry`. Or the page class would need to know when the `Title` and `Text` properties of the `Note` object changed values.

The basic problem involves the existence of properties that change value without notifying anybody of the change. Can we do something about this?

Version 4. I will notify you when the property changes

Let's step back a moment.

So far, all the versions of the program have contained a class deriving from `ContentPage` that displays a user interface allowing a user to enter and edit two pieces of text. These two pieces of text are also stored in a class named `Note`. This `Note` class stores data that underlies the user interface of the page class.

These two classes are really two sides of the same data—one class presents the data for editing by the user, and the other class handles the more low-level chores, including loading and saving the data in the file system.

Optimally, at any time, both classes should be dealing with the same data. But this is not the case. The page class doesn't know when the data in the `Note` class has changed, and the `Note` class doesn't know when the text in the `Entry` and `Editor` views has changed.

Keeping user interfaces in synchronization with underlying data is a common problem, and standard solutions are available to fix that problem. One of the most important is an interface named `INotifyPropertyChanged`. It's defined in the .NET `System.ComponentModel` namespace like so:

```
interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

The entire interface consists of just one event named `PropertyChanged`, but this event provides a simple universal way for a class to notify any other class that might be interested when one of its properties has changed values.

The PropertyChangedEventHandler delegate associated with the PropertyChanged event incorporates an event argument of PropertyChangedEventArgs. This class defines a public property of type string named PropertyName that identifies the property being changed.

What's that? A property named PropertyName that identifies the property being changed? Yes, it sounds a little confusing, but in practice it's quite simple.

The following **NoteTaker4** program was created with the PCL template, but a Shared Asset Project could implement these changes as well.

A class such as Note can implement the INotifyPropertyChanged interface by simply indicating that the class derives from that interface and including a public event of the correct type and name:

```
class Note : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    …

}
```

In theory, that's all that's required. However, a class that implements this interface should also actually fire the event whenever one of its public properties changes value. The PropertyChanged-EventArgs object accompanying the event identifies the actual property that's changed value. The property should have been assigned its new value by the time it fires the event.

In the previous versions of the Note class, the properties were defined with implicit backing fields:

```
public string Title { set; get; }

public string Text { set; get; }
```

Now they're going to need explicit private backing fields:

```
string title, text;
```

Here's the new definition of the Title property. The Text property is similar:

```
public string Title
{
    set
    {
        if (title != value)
        {
            title = value;

            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs("Title"));
            }
        }
    }
    get
    {
        return title;
    }
}
```

This is very standard `INotifyPropertyChanged` code. The `set` accessor begins by checking if the private field is the same as the incoming string, and only continues if it's not. Some programmers new to `INotifyPropertyChanged` want to skip this check, but it's important. The interface is called `INotifyPropertyChanged` and not `INotifyMaybePropertyChangedMaybeNot`. In some cases, failing to check if the property is actually changing can cause infinite recursion.

The `set` accessor continues by saving the new value in the backing field and only then firing the event.

Here's the complete `Note` class:

```
using System.ComponentModel;

namespace NoteTaker4
{
    class Note : INotifyPropertyChanged
    {
        string title, text;

        public event PropertyChangedEventHandler PropertyChanged;

        public string Title
        {
            set
            {
                if (title != value)
                {
                    title = value;

                    if (PropertyChanged != null)
                    {
                        PropertyChanged(this, new PropertyChangedEventArgs("Title"));
                    }
                }
            }
            get
            {
                return title;
            }
        }

        public string Text
        {
            set
            {
                if (text != value)
                {
                    text = value;

                    if (PropertyChanged != null)
                    {
                        PropertyChanged(this, new PropertyChangedEventArgs("Text"));
                    }
                }
            }
            get
            {
                return text;
            }
        }

        public void Save(string filename)
        {
            string text = this.Title + "\n" + this.Text;
            FileHelper.WriteAllText(filename, text, () => { });
        }

        public void Load(string filename)
        {
            FileHelper.ReadAllText(filename, (string text) =>
                {
                    // Break string into Title and Text.
                    int index = text.IndexOf('\n');
                    this.Title = text.Substring(0, index);
                    this.Text = text.Substring(index + 1);
                });
        }
    }
}
```

The various `FileHelper` classes are the same as those in **NoteTaker3Pcl**.

The `NoteTaker4Page` class defines an instance of `Note` as a field (as in the previous version of the program), but now the constructor also attaches a handler for the `PropertyChanged` event now defined by `Note`:

```
note.PropertyChanged += (sender, args) =>
    {
        switch (args.PropertyName)
        {
            case "Title":
                entry.Text = note.Title;
                break;

            case "Text":
                entry.Text = note.Text;
                break;
        }
    };
```

This could be a named event handler of course, and it could use an `if` and `else` rather that a `switch` and `case` to identify the property being changed. It then sets the new value of the property to the `Text` property of either the `Entry` or `Editor`.

It looks fine, but it still won't work on the Windows Phone. When you tap the **Load** button you'll get an Unauthorized Access exception. Now what's wrong?

Here's the problem: In the general case, callbacks from asynchronous methods do *not* execute in the same thread as the code that the initiated the operation. Instead, the callback executes in the background thread that carried out the asynchronous operation.

Let's follow it through: When you press the **Load** button, the Windows Phone `ReadAllText` method executes. When the text is obtained, it calls the `completed` method but in a secondary thread of execution. In the `Load` method in `Note`, that `completed` method sets the `Title` and `Text` properties. The new `Title` property causes a `PropertyChanged` event to fire, and in that handler the new `Title` property is set to the `Text` property of the `Entry` view.

Therefore, the `Entry` view is being accessed from a thread other than the user-interface thread, and that's not allowed. That's why the exception is raised.

Fortunately, the fix for this problem is fairly easy. The `Device` class has a `BeginInvokeOn-MainThread` method with an argument of type `Action`. Simply enclose the code you want to execute in the UI thread in the body of that `Action` argument. It's easiest to wrap the entire `switch` and `case` in that callback:

```
note.PropertyChanged += (sender, args) =>
    {
        Device.BeginInvokeOnMainThread(() =>
            {
                switch (args.PropertyName)
                {
                    case "Title":
                        entry.Text = note.Title;
                        break;

                    case "Text":
                        editor.Text = note.Text;
                        break;
                }
            });
    };
```

The `Device.BeginInvokeOnMainThread` effectively waits until the UI thread gets a time slice from the operating system's thread scheduler, and then it runs the specified code.

With this change, you'll find that when you rerun the Windows Phone app, you can press **Load** just once and the `Entry` and `Editor` will be set with the saved values. They're being set not in the handler for the **Load** button but in the `PropertyChanged` handler when the properties are actually updated with the values loaded from the file.

You can also go the other way and keep the `Note` class updated with the current values of the `Entry` and `Editor` views. Simply install `TextChanged` handlers:

```
entry.TextChanged += (sender, args) =>
    {
        note.Title = args.NewTextValue;
    };

editor.TextChanged += (sender, args) =>
    {
        note.Text = args.NewTextValue;
    };
```

Wait a minute. Have we messed this up? The `PropertyChanged` handler is setting the `Entry` and `Editor` text from the `Note` properties, and now these two `TextChanged` handlers are setting the `Note` properties from the `Entry` and `Editor` text. Isn't that an infinite loop?

No, because `Entry`, `Editor`, and `Note` fire `Changed` events only when the property is actually changing. The potentially infinite loop is truncated when the corresponding properties are the same.

Now that the `Entry` and `Editor` views are kept consistent with the `Note` class, it's not necessary to set the `Note` object from the `Entry` and `Editor` in the `Save` handler. Nor do we need to set the `Entry` and `Editor` from the `Note` object in the `Load` handler. Here's the complete `NoteTaker4-Page` class. Notice that the `Entry` and `Editor` instances no longer need to be saved as fields because they're no longer referenced in the `Clicked` handlers:

```
class NoteTaker4Page : ContentPage
{
    static readonly string FILENAME = "test.note";

    Note note = new Note();
    Button loadButton;

    public NoteTaker4Page()
    {
        // Create Entry and Editor views.
        Entry entry = new Entry
        {
            Placeholder = "Title (optional)"
        };

        Editor editor = new Editor
        {
            Keyboard = Keyboard.Create(KeyboardFlags.All),
            BackgroundColor = Device.OnPlatform(Color.Default,
                                                Color.Default,
                                                Color.White),
            VerticalOptions = LayoutOptions.FillAndExpand
        };

        // Create Save and Load buttons.
        Button saveButton = new Button
```

```
{
    Text = "Save",
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
saveButton.Clicked += OnSaveButtonClicked;

loadButton = new Button
{
    Text = "Load",
    IsEnabled = false,
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
loadButton.Clicked += OnLoadButtonClicked;

// Check if the file is available.
FileHelper.Exists(FILENAME, (exists) =>
    {
        loadButton.IsEnabled = exists;
    });

// Handle the Note's PropertyChanged event.
note.PropertyChanged += (sender, args) =>
{
    Device.BeginInvokeOnMainThread(() =>
        {
            switch (args.PropertyName)
            {
                case "Title":
                    entry.Text = note.Title;
                    break;

                case "Text":
                    editor.Text = note.Text;
                    break;
            }
        });
};

// Handle the Entry and Editor TextChanged events.
entry.TextChanged += (sender, args) =>
    {
        note.Title = args.NewTextValue;
    };

editor.TextChanged += (sender, args) =>
    {
        note.Text = args.NewTextValue;
    };

// Assemble page.
this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);

this.Content = new StackLayout
{
```

```
            Children =
            {
                new Label
                {
                    Text = "Title:"
                },
                entry,
                new Label
                {
                    Text = "Note:"
                },
                editor,
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    Children =
                    {
                        saveButton,
                        loadButton
                    }
                }
            }
        };
    }

    void OnSaveButtonClicked(object sender, EventArgs args)
    {
        note.Save(FILENAME);
        loadButton.IsEnabled = true;
    }

    void OnLoadButtonClicked(object sender, EventArgs args)
    {
        note.Load(FILENAME);
    }
}
```

As you test this new version, you might want to restore the phone or simulator to a state where no file has yet been saved. You can do that simply by uninstalling the application from the phone or simulator. That uninstall removes all the data stored along with the application as well.

Version 5. Data binding

Xamarin.Forms is mostly about user interfaces, but user interfaces rarely exist in isolation. A case in point is the application being built in this chapter. This user interface is really a visual representation of data and the means through which that data is manipulated.

To accommodate such scenarios, .NET and Xamarin.Forms provide some built-in facilities for smoothing the links between data and user interfaces. The **NoteTaker5** program incorporates some of these features. In particular, you'll see here how to automate the process of data binding: linking two properties of two objects so that changes to one property automatically trigger a change to the other.

Streamlining INotifyPropertyChanged classes

Classes that implement `INotifyPropertyChanged` usually have rather more changeable properties than the `Note` class. For this reason, it's a good idea to simplify the `set` accessors, if only to avoid mixing up property and field names, or misspelling the text property name.

One simplification is to encapsulate the actual firing of the event in a protected virtual method:

```
protected virtual void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

Now the `Title` property looks like this:

```
public string Title
{
    set
    {
        if (title != value)
        {
            title = value;
            OnPropertyChanged("Title");
        }
    }
    get
    {
        return title;
    }
}
```

The `OnPropertyChanged` method is made protected and virtual because you might write an enhanced `Note` class that derives from this `Note` class but includes more properties. The derived class needs access to this method.

But let's pause a moment to improve out `INotifyPropertyChanged` handling. It's recommended to obtain the handler first, and then perform the check for `null` on and the event call on that same object:

```
protected virtual void OnPropertyChanged(string propertyName)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
        handler(this, new PropertyChangedEventArgs(propertyName));
}
```

In a multithreaded environment, a PropertyChanged handler might be detached between the null check and the call, and this code prevents a null-reference exception from occurring.

You can go further in streamlining the `OnPropertyChanged` method. C# 5.0 introduced support for `CallerMemberNameAttribute` and some related attributes. This attribute allows you to replace an optional method argument with the name of the calling method or property.

In the `OnPropertyChanged` method, make the argument optional by assigning `null` to it and precede it with `CallerMemberName` in square brackets:

```
protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
        handler(this, new PropertyChangedEventArgs(propertyName));
}
```

You'll need a `using` directive for `System.Runtime.CompilerServices` for that attribute. Now the `Title` property can call `OnPropertyChanged` with no arguments, and the `propertyName` argument will automatically be set to the property name "Title" because that's where the call to `OnPropertyChanged` is originating:

```
public string Title
{
    set
    {
        if (title != value)
        {
            title = value;
            OnPropertyChanged();
        }
    }
    get
    {
        return title;
    }
}
```

This avoids a potentially misspelled text property name, and allows property names to be changed during program development without worrying about also changing text strings. One of the primary reasons the `CallerMemberName` was invented was to simplify classes that implement `INotify-PropertyChanged`.

It's possible to go even further. You'll need to define a generic method named `SetProperty` (for example) with the `CallerMemberName` attribute, but you'll need to remove it from `OnProperty-Changed`:

```
bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
{
    if (Object.Equals(storage, value))
        return false;

    storage = value;
    OnPropertyChanged(propertyName);
    return true;
}

protected virtual void OnPropertyChanged(string propertyName)
{
    PropertyChangedEventHandler handler = PropertyChanged;
    if (handler != null)
        handler(this, new PropertyChangedEventArgs(propertyName));
}
```

The `SetProperty` method requires access to the backing field and the new value, but automates the rest of the process and returns `true` if the property was changed. (You might need to use this return value if you're doing some additional processing in the `set` accessor.) Now the `Title` property looks like this:

```
public string Title
{
    set
    {
        SetProperty(ref title, value);
    }
    get
    {
        return title;
    }
}
```

Although `SetProperty` is a generic method, the C# compiler can deduce the type from the arguments. The whole property definition has become so short, you can write the accessors concisely on single lines without obscuring the operations:

```
public string Title
{
    set { SetProperty(ref title, value); }
    get { return title; }
}
```

Here is the new `Note` class in the PCL project **NoteTaker5**:

```
class Note : INotifyPropertyChanged
{
    string title, text;

    public event PropertyChangedEventHandler PropertyChanged;

    public string Title
    {
        set { SetProperty(ref title, value); }
        get { return title; }
    }

    public string Text
    {
        set { SetProperty(ref text, value); }
        get { return text; }
    }

    public void Save(string filename)
    {
        string text = this.Title + "\n" + this.Text;
        FileHelper.WriteAllText(filename, text, () => { });
    }

    public void Load(string filename)
    {
        FileHelper.ReadAllText(filename, (string text) =>
            {
                // Break string into Title and Text.
                int index = text.IndexOf('\n');
                this.Title = text.Substring(0, index);
                this.Text = text.Substring(index + 1);
            });
    }

    bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
    {
        if (Object.Equals(storage, value))
            return false;

        storage = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    protected virtual void OnPropertyChanged(string propertyName)
    {
        PropertyChangedEventHandler handler = PropertyChanged;
        if (handler != null)
            handler(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

This type of streamlining obviously makes much more sense for classes with more than just two properties, but then it begins making lots of sense.

A peek into BindableObject and bindable properties

You've seen how you can define a class that implements the `INotifyPropertyChanged` interface. You'll probably be interested to learn that many of the classes in Xamarin.Forms—including all the view, layout, and page classes—also implement `INotifyPropertyChanged`. All these classes have a `PropertyChanged` event that your applications can use to be notified when properties of these classes change.

For example, the **NoteTaker4** program keeps the `Title` property of the `Note` class updated from the `Text` property of the `Entry` view like so:

```
entry.TextChanged += (sender, args) =>
    {
        note.Title = args.NewTextValue;
    };
```

You can do pretty much the same thing by installing a handler for the `PropertyChanged` event of the `Entry` view and checking for the `Text` property:

```
entry.PropertyChanged += (sender, args) =>
    {
        if (args.PropertyName == "Text")
            note.Title = entry.Text;
    };
```

In a sense, the `TextChanged` event defined by `Entry` and `Editor` is redundant and unnecessary. It is provided solely for purposes of programmer convenience.

Many of the classes in Xamarin.Forms implement `INotifyPropertyChanged` automatically because they derive from a class named `BindableObject` that implements this interface. `BindableObject` also defines a protected virtual method named `OnPropertyChanged`.

As its name implies, `BindableObject` is important to the support of data binding in Xamarin.-Forms, yet the implementation of `INotifyPropertyChanged` by this class is only part of the story and, to be honest, the easier part.

Let's begin exploring the more arcane part of `BindableObject` with some experimentation. In one of the previous versions of the **NoteTaker** program, try initializing the `Entry` view with some text:

```
entry.Text = "This is some text";
```

Now when you run the program the `Entry` is initialized with this text. But now try replacing that property setting with this method call:

```
entry.SetValue(Entry.TextProperty, "This is some text");
```

This works as well. These two statements are functionally identical.

Look at that weird first argument to `SetValue`: It's something called `Entry.TextProperty`, which indicates that it's static, but it's not a property at all. It's static *field* of the `Entry` class. It's also read-only, and it's defined in the `Entry` class something like this:

```
public static readonly BindableProperty TextProperty;
```

It's a little odd for a field of a class to be named `TextProperty`, but there it is. Because it's static, however, it exists independently of any `Entry` objects that might or might not exist.

If you look in the documentation of the `Entry` class, you'll see that it defines four properties—`Text`, `TextColor`, `IsPassword`, and `Placeholder`—and you'll also see four corresponding public static read-only fields of type `BindableProperty` with the names `TextProperty`, `TextColor-Property`, `IsPasswordProperty`, and `PlaceholderProperty`.

These properties and fields are closely related. Indeed, internal to the Entry class, the Text property is defined like this:

```
public string Text
{
    set { SetValue(Entry.TextProperty, value); }
    get { return (string)GetValue(Entry.TextProperty); }
}
```

So you see why it is that your application calling SetValue on Entry.TextProperty is exactly equivalent to setting the Text property and perhaps just a tinier bit faster!

The internal definition of the Text property in Entry isn't secret information. This is standard code. The SetValue and GetValue methods are defined by BindableObject, the same class that implements INotifyPropertyChanged for many Xamarin.Forms classes. All the real work involved with maintaining the Text property is going on in these SetValue and GetValue calls. Casting is required for the GetValue method because it's defined as returning object.

The static Entry.TextProperty object is of type BindableProperty, which you might correctly surmise is a class related to BindableObject, but it's important to keep them distinct in your mind: BindableObject is the class from which many Xamarin.Forms classes derive and that provides support for objects of type BindableProperty.

The BindableProperty objects effectively extend the functionality of standard C# properties. Bindable properties provide systematic ways to:

- Define properties
- Give properties default values
- Store their current values
- Provide mechanisms for validating property values
- Maintain consistency among related properties in a single class
- Respond to property changes
- Trigger notifications when a property is about to change and has changed

In addition, BindableObject and BindableProperty provide mechanisms for animation and data binding. They are a vital part of the infrastructure of Xamarin.Florms.

The close relationship of a property named Text with a BindableProperty named Text-Property is reflected in the way that programmers speak about these properties: Sometimes a programmer says that the Text property is "backed by" a BindableProperty named TextProp-erty because TextProperty provides infrastructure support for the Text property. But a common shortcut is to say that Text is itself a "bindable property" and generally no one will be confused.

Not every Xamarin.Forms property is a bindable property. Neither the Content property of

ContentPage nor the Children property of Layout<T> (from which StackLayout derives) is a bindable property. Sometimes changes to nonbindable properties result in the PropertyChanged event being fired, but that's not guaranteed. The PropertyChanged event is only guaranteed for bindable properties.

Later in this book you'll see how to define your own bindable properties.

Automated data bindings

As you've seen, it's possible to install event handlers on the Entry and Editor to determine when the Text property changes, and to use that occasion to set the Title and Text properties of the Note class. Similarly, you can use the PropertyChanged event of the Note class to keep the Entry and Editor updated.

In other words, the **NoteTaker4** program has paired up objects and properties so that they track each other's values. As one property changes, the other is updated.

It turns out that tasks like this are very common, and this is why Xamarin.Forms allows such tasks to be automated with a technique called *data binding*.

Data bindings involve a source and a target. The source is the object and property that changes, and the target is the object and property that is changed as a result. But that's a simplification. Although the distinction between target and source is clearly defined in any particular data binding, sometimes the properties affect each other in different ways: Sometimes the target causes the source to be updated, and sometimes the source and target update each other.

The data binding mechanism in Xamarin.Forms uses the PropertyChanged event to determine when a source property has changed. Therefore, a source property used in a data binding must be part of a class that implements INotifyPropertyChanged (either directly or through inheritance), and the class must fire a PropertyChanged event when that property changes. (Actually this is not entirely true: If the source property never changes, a PropertyChanged event is *not* required for the data binding to work, but it's only a "one time" data binding and not very interesting.)

The target property of a data binding must be backed by a BindableProperty. As you'll see, this requirement is imposed by the programming interface for data bindings. You cannot set a data binding without referencing a BindableProperty object!

In the **NoteTaker5** program, the Text properties of Entry and Editor are the binding targets because they are backed by bindable properties named TextProperty. The Title and Text properties of the Note class are the binding sources. The Note class implements INotifyProperty-0Changed so PropertyChanged events are fired when these source properties changes.

You can define a data binding to keep a target property updated with the value of a source property with two statements: The first statement associates the two objects by setting the BindingContext property of the target object to the source object. In this case that's the instance of the Note class:

```
entry.BindingContext = note;
```

The second statement makes use of a SetBinding method on the target. These SetBinding calls come in several different forms. BindableObject itself defines one SetBinding method, and the BindableObjectExtensions class defines two SetBinding extension methods. Here's the simplest:

```
entry.SetBinding(Entry.TextProperty, "Title");
```

You'll see more complex data bindings in the chapters ahead. The target property here is the Text property of Entry. That's specified in the first argument. The Title property—specified here as a text string—is assumed to be a property of whichever object has been defined as the BindingContext of the Entry object, which in this case is a Note object.

Similarly, you can set a data binding for the Editor:

```
editor.BindingContext = note;
editor.SetBinding(Editor.TextProperty, "Text");
```

The first argument of SetBinding must be a BindableProperty object defined by the target class (Editor in this case) or inherited by the target class.

Part of the infrastructure that BindableProperty provides is a default binding mode that defines the relationship between the source and the target. The BindingMode enumeration has four members:

- Default
- OneWay — source updates target, the normal case
- OneWayToSource — source updates target
- TwoWay — source and target update each other

For the TextProperty objects defined by Entry and Editor, the default binding mode is BindingMode.TwoWay. This means that these four statements actually define a two-way data binding: Any change to the Title property of the Note object is reflected in the Text property of the Entry, and vice versa, and the same goes for the Editor.

The **NoteTaker4** program included three event handlers—the PropertyChanged handler for the Note class and the TextChanged handlers for the Entry and Editor—to keep these pairs of properties in synchronization. Those three event handlers are no longer required if they are replaced by the four statements you've just seen:

```
entry.BindingContext = note;
entry.SetBinding(Entry.TextProperty, "Title");

editor.BindingContext = note;
editor.SetBinding(Editor.TextProperty, "Text");
```

Internally, these four statements result in similar event handlers being set. That's how the data-binding mechanism works.

However, these four statements can actually be reduced to three statements. Here's how:

The `BindingContext` property has a very special characteristic. It is very likely that more than one data binding on a page has the same `BindingContext`. That is true for this little example. For this reason, the `BindingContext` property is propagated through the visual tree of a page. In other words, if you set the `BindingContext` on a page, it will propagate to all the views on that page except for those views that have their own `BindingContext` properties set to something else. You can set `BindingContext` on a `StackLayout` and it will propagate to all the children (and other descendants) of that `StackLayout`. The two `BindingContext` settings shown above can be replaced with one set on the page itself:

```
this.BindingContext = note;
```

These automated data bindings are part of the **NoteTaker5Page** class. Also, the handlers for the **Load** and **Save** buttons have become lambda functions, all the variables have been moved to the constructor, and the code is looking quite sleek at this point:

```
class NoteTaker5Page : ContentPage
{
    static readonly string FILENAME = "test.note";

    public NoteTaker5Page()
    {
        // Create Entry and Editor views.
        Entry entry = new Entry
        {
            Placeholder = "Title (optional)"
        };

        Editor editor = new Editor
        {
            Keyboard = Keyboard.Create(KeyboardFlags.All),
            BackgroundColor = Device.OnPlatform(Color.Default,
                                                Color.Default,
                                                Color.White),
            VerticalOptions = LayoutOptions.FillAndExpand
        };

        // Set data bindings.
        Note note = new Note();
        this.BindingContext = note;
        entry.SetBinding(Entry.TextProperty, "Title");
        editor.SetBinding(Editor.TextProperty, "Text");

        // Create Save and Load buttons.
        Button saveButton = new Button
        {
            Text = "Save",
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };

        Button loadButton = new Button
        {
            Text = "Load",
            IsEnabled = false,
            HorizontalOptions = LayoutOptions.CenterAndExpand
        };

        // Set Clicked handlers.
        saveButton.Clicked += (sender, args) =>
            {
                note.Save(FILENAME);
                loadButton.IsEnabled = true;
            };

        loadButton.Clicked += (sender, args) => note.Load(FILENAME);

        // Check if the file is available.
        FileHelper.Exists(FILENAME, (exists) =>
            {
                loadButton.IsEnabled = exists;
            });

        // Assemble page.
        this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);

        this.Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "Title:"
                },
                entry,
                new Label
                {
                    Text = "Note:"
                },
                editor,
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    Children =
                    {
                        saveButton,
                        loadButton
                    }
                }
            }
        };
    }
}
```

By replacing the explicit event handlers with data bindings, we've also managed to forget all about the problem of the `PropertyChanged` event in the `Note` class being fired from a secondary thread. When you use a data binding, you don't have to worry about that.

Eventually, the final **NoteTaker** application won't be dealing with a single `Note` object. It will have a whole collection of `Note` objects. With these data bindings in place, you can view and edit any one of these `Note` objects by setting it to the `BindingContext` of the page. But that's for the next chapter.

Meanwhile, let's see if we can simplify this page code even more. One prime candidate is the code that enables the **Load** button based on the `FileHelper.Exists` call. Perhaps solving that problem will simplify some other parts of the program as well.

Version 6. Awaiting results

As you've seen, asynchronous operations can be difficult to manage. Sometimes code executes in a different order than you anticipated and some thought is required to figure out what's going on. Asynchronous programming will likely never be quite as simple as single-threaded coding, but some of the difficulty in working with asynchronous operations has been alleviated with C# 5.0, released in 2012. C# 5.0 introduced a revolutionary change in the way that programmers deal with asynchronous operations. This change consists of a keyword named `async` and an operator named `await`.

The `async` keyword is mostly for purposes of backward compatibility. The `await` operator is the big one. It allows programmers to work with asynchronous functions almost as if they were relatively normal imperative programming statements without callback methods.

Let's look at a generalized use of an asynchronous method that might appear in a Windows Phone 8 program. You have a method in your program that calls an asynchronous method in the operating system and sets a `Completed` handler implemented as a separate method. The comments indicate some other code that might appear in the method:

```
void SomeMethod()
{
    // Often some initialization code
    IAsyncOperation<SomeType> asyncOp = sysClass.LongJobAsync(…);
    // Some additional code
    asyncOp.Completed = MyCompletedHandler;
    // And perhaps still more code
}
```

The statement that sets the `Completed` property returns quickly while the actual asynchronous operation goes on in the background. All the code in `SomeMethod` will execute before the callback method is called. Here's what the `Completed` handler might look like if you chose to ignore cancellations or errors encountered in the background process:

```
void MyCompletedHandler(IAsyncOperation<SomeType> op, AsyncStatus status)
```

```
{
    SomeType myResult = op.GetResults();
    // Code using the result of the asynchronous operation
}
```

The handler can also be written as a lambda function.

```
void SomeMethod()
{
    // Often some initialization code
    IAsyncOperation<SomeType> asyncOp = sysClass.LongJobAsync(…);
    // Some additional code
    asyncOp.Completed = (op, status) =>
    {
        SomeType myResult = op.GetResults();
        // Code using the result of the asynchronous operation
    };
    // And perhaps still more code
}
```

The additional code indicated with the comment "And perhaps still more code" will execute and `SomeMethod` will return before the code in the `Completed` handler executes. The order that the code appears in the method is not the same order that the code executes, and this is one reason why using lambda functions for asynchronous operations can sometimes be a bit confusing.

Here's how `SomeMethod` can be rewritten using the `await` operator:

```
async void SomeMethod()
{
    // Often some initialization code
    IAsyncOperation<SomeType> asyncOp = sysClass.LongJobAsync(…);
    // Some additional code
    // And perhaps still more code
    SomeType myResult = await asyncOp;
    // Code using the result of the asynchronous operation
}
```

Notice that `SomeMethod` now includes the `async` modifier. This is required for backward compatibility. In versions of C# prior to 5.0, `await` was not a keyword so it could be used as a variable name. To prevent C# 5.0 from breaking that code, the `async` modifier is required to indicate a method that includes `await`.

The `Completed` handler still exists in this code, but it's not exactly obvious where it is. It's basically everything to the left of the `await` operator, and everything below the statement containing the `await` operator.

The C# compiler performs the magic. The compiler recognizes `IAsyncOperation` as encapsulating an asynchronous method call and basically turns `SomeMethod` into a state machine. The method executes normally up until the `await` operator, and then the method returns. At this point, the background process is running and other code in the program can run as well. When the background process is completed, execution of the method resumes with the assignment to the `myResult` variable.

If you can organize your code to eliminate the two comments labeled as “Some additional code” and “And perhaps still more code” you can skip the assignment to the IAsyncOperation object and just get the result:

```
async void SomeMethod()
{
    // Often some initialization code
    SomeType myResult = await sysClass.LongJobAsync(…);
    // Code using the result of the asynchronous operation
}
```

This is how await is customarily used—as an operator between a method that runs asynchronously and the assignment statement to save the result.

Don’t let the name of the await operator fool you! The code doesn’t actually sit there waiting. The SomeMethod method returns and the processor is free to run other code. Only when the asynchronous operation has completed does the code in SomeMethod resume execution where it left off.

Of course, any time we’re dealing with file I/O, we should be ready for problems. What happens if the file isn’t there, or you run out of storage space? Even when problems don’t occur, sometimes error results are normal: The Windows Phone version of the Exists method uses an error reported in the Completed handler to determine if the file exists or not. So how are errors handled with await?

If you use await with an asynchronous operation that encounters an error or is cancelled, await throws an exception. If you need to handle errors or cancellations, you can put the await operator in a try and catch block, looking something like this:

```
SomeType myresult = null;
try
{
    myresult = await sysClass.LongJobAsync(…);
}
catch (OperationCanceledException)
{
    // handle a cancellation
}
catch (Exception exc)
{
    // handle an error
}
```

Now let’s use await in some real code. Here’s the Windows Phone version of the old ReadAllText method in **NoteTaker5**:

```
public void ReadAllText(string filename, Action<string> completed)
{
    StorageFolder localFolder = ApplicationData.Current.LocalFolder;
    IAsyncOperation<StorageFile> createOp = localFolder.GetFileAsync(filename);
    createOp.Completed = (asyncInfo1, asyncStatus1) =>
    {
        IStorageFile storageFile = asyncInfo1.GetResults();
        IAsyncOperation<IRandomAccessStreamWithContentType> openOp =
                                storageFile.OpenReadAsync();
        openOp.Completed = (asyncInfo2, asyncStatus2) =>
        {
            IRandomAccessStream stream = asyncInfo2.GetResults();
            DataReader dataReader = new DataReader(stream);
            uint length = (uint)stream.Size;
            DataReaderLoadOperation loadOp = dataReader.LoadAsync(length);
            loadOp.Completed = (asyncInfo3, asyncStatus3) =>
            {
                string text = dataReader.ReadString(length);
                dataReader.Dispose();
                completed(text);
            };
        };
    };
}
```

This has three nested `Completed` handlers. Because the method returns before even the first `Completed` handler executes, it is not possible to return the contents of the file from the method. The file contents must instead be returned through a function passed to the method.

Here's how it can be rewritten using `await`:

```
public async void ReadAllText(string filename, Action<string> completed)
{
    StorageFolder localFolder = ApplicationData.Current.LocalFolder;
    IStorageFile storageFile = await localFolder.GetFileAsync(filename);
    IRandomAccessStream stream = await storageFile.OpenReadAsync();
    DataReader dataReader = new DataReader(stream);
    uint length = (uint)stream.Size;
    await dataReader.LoadAsync(length);
    string text = dataReader.ReadString(length);
    dataReader.Dispose();
    completed(text);
}
```

Notice the `async` modifier on the method. The `async` modifier does not change the signature of the method, so this method is still considered a proper implementation of the `ReadAllText` method in the `IFileHelper` interface.

The `await` operator appears three times; the first two times on methods that return an object and the third time on the `LoadAsync` method that just performs an operation without returning anything.

Behind the scenes, the C# compiler divides this method into four chunks of execution. The method returns to the caller (the `Note` class) at the first `await` operator and then resumes when the `GetFileAsync` method has completed, leaving again at the next `await`, and so on.

Actually, it's possible for the `ReadAllText` method to execute sequentially from start to finish. If the asynchronous methods complete their operations before the `await` operator is evaluated, execution just continues as if it were a normal function call. This is a performance optimization that often plays a role in the relatively fast solid state file I/O on mobile devices.

Let's improve this ReadAllText method a bit. The DataReader class implements the IDisposable interface and includes a Dispose method. Failing to call Dispose can leave the file open. Calling Dispose also closes the IRandomAccessStream on which the DataReader is based. IRandomAccessStream also implements IDisposable.

It's a good idea to enclose IDisposable objects in using blocks. This ensures that the Dispose method is automatically called even if an exception is thrown inside the block. A better implementation of ReadAllText is this:

```
public async void ReadAllText(string filename, Action<string> completed)
{
    StorageFolder localFolder = ApplicationData.Current.LocalFolder;
    IStorageFile storageFile = await localFolder.GetFileAsync(filename);
    using (IRandomAccessStream stream = await storageFile.OpenReadAsync())
    {
        using (DataReader dataReader = new DataReader(stream))
        {
            uint length = (uint)stream.Size;
            await dataReader.LoadAsync(length);
            string text = dataReader.ReadString(length);
            completed(text);
        }
    }
}
```

Now the explicit call to Dispose is not required.

Of course, we still have the annoyance of passing a callback function to the ReadAllText method. But what's the alternative? If the method actually returns to the caller at the first await operator, how can the method return the text contents of the file? Trust the C# compiler. If it can implement await, surely it can also allow you to create your own asynchronous methods.

Let's replace ReadAllText with a method named ReadTextAsync. The new name reflects the fact that this method is itself asynchronous and can be called with the await operator to return a string with the contents of the file.

To do this, the ReadTextAsync method needs to return a Task object. The Task class is defined in System.Threading.Tasks and is the standard .NET representation of an asynchronous operation. The Task class is quite extensive, but only a little bit of it is necessary in this context. The System.Threading.Tasks namespace actually defines two Task classes:

- Task for asynchronous methods that return nothing
- Task<TResult> for asynchronous methods that return an object of type TResult.

These are the .NET equivalences of the Windows 8 IAsyncAction and IAsyncOperation-<TResult> interfaces, and there are extension methods that convert the Task objects to IAsyncAction and IAsyncOperation<TResult> objects.

Instead of returning `void`, this new method can return `Task<string>`. The callback argument isn't required, and inside the method, the file's contents are returned as if this were a normal method:

```
public async Task<string> ReadTextAsync(string filename)
{
    StorageFolder localFolder = ApplicationData.Current.LocalFolder;
    IStorageFile storageFile = await localFolder.GetFileAsync(filename);
    using (IRandomAccessStream stream = await storageFile.OpenReadAsync())
    {
        using (DataReader dataReader = new DataReader(stream))
        {
            uint length = (uint)stream.Size;
            await dataReader.LoadAsync(length);
            return dataReader.ReadString(length);
        }
    }
}
```

The compiler handles the rest. The `return` statement seems to return a `string` object, which is the return value of the `ReadString` method, but the C# compiler automatically wraps that value in a `Task<string>` object actually returned from the method at the execution of the first `await` operator.

This `ReadTextAsync` method can now be called using an `await` operator.

Of course, redefining the method signature of these file I/O functions has an impact throughout the program. If we want to continue to use `DependencyService` to call platform-independent methods—and that is highly desirable—the iOS and Android methods should have the same signature. This means that the **NoteTaker6** version of `IFileHelper` consists of these three asynchronous methods:

```
using System.Threading.Tasks;

namespace NoteTaker6
{
    public interface IFileHelper
    {
        Task<bool> ExistsAsync(string filename);

        Task WriteTextAsync(string filename, string text);

        Task<string> ReadTextAsync(string filename);
    }
}
```

There are no longer any callback functions in the methods. The `Task` object has its own callback mechanism.

Here's the complete Windows Phone version of the `FileHelper` implementation of `IFileHelper`:

```
using System;
using System.Threading.Tasks;
using Windows.Storage;
using Windows.Storage.Streams;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker6.WinPhone.FileHelper))]

namespace NoteTaker6.WinPhone
{
    class FileHelper : IFileHelper
    {
        public async Task<bool> ExistsAsync(string filename)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;

            try
            {
                await localFolder.GetFileAsync(filename);
            }
            catch
            {
                return false;
            }
            return true;
        }

        public async Task WriteTextAsync(string filename, string text)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IStorageFile storageFile =
                    await localFolder.CreateFileAsync(filename,
                                CreationCollisionOption.ReplaceExisting);

            using (IRandomAccessStream stream =
                    await storageFile.OpenAsync(FileAccessMode.ReadWrite))
            {
                using (DataWriter dataWriter = new DataWriter(stream))
                {
                    dataWriter.WriteString(text);
                    await dataWriter.StoreAsync();
                }
            }
        }

        public async Task<string> ReadTextAsync(string filename)
        {
            StorageFolder localFolder = ApplicationData.Current.LocalFolder;
            IStorageFile storageFile = await localFolder.GetFileAsync(filename);
            using (IRandomAccessStream stream = await storageFile.OpenReadAsync())
            {
                using (DataReader dataReader = new DataReader(stream))
                {
                    uint length = (uint)stream.Size;
                    await dataReader.LoadAsync(length);
                    return dataReader.ReadString(length);
                }
            }
        }
    }
}
```

Notice the use of the `try` and `catch` block on the `await` operation in the `ExistsAsync` method.

The `WriteTextAsync` method doesn't return a value, and the return value of the method is simply `Task`. Such a method doesn't require an explicit `return` statement. A method that returns `Task<TResult>` needs a `return` statement with a `TResult` object. In either case, all the asynchronous calls in the method should be preceded with `await`. (There are alternatives but they are somewhat more complicated. Asynchronous methods without any `await` operators can also be handled somewhat differently as you'll see shortly.)

For the iOS and Android versions, the methods now need to return `Task` and `Task<TResult>` objects, but up to this point the methods themselves haven't been asynchronous. One solution is to switch to using asynchronous file I/O, at least in part. Many of the I/O methods in `System.IO` have asynchronous versions. That's what's been done here with the `WriteTextAsync` and `ReadText-Async` methods:

```
using System;
using System.IO;
using System.Threading.Tasks;
using Xamarin.Forms;

[assembly: Dependency(typeof(NoteTaker6.iOS.FileHelper))]

namespace NoteTaker6.iOS
{
    class FileHelper : IFileHelper
    {
        public Task<bool> ExistsAsync(string filename)
        {
            string filepath = GetFilePath(filename);
            bool exists = File.Exists(filepath);
            return Task<bool>.FromResult(exists);
        }

        public async Task WriteTextAsync(string filename, string text)
        {
            string filepath = GetFilePath(filename);
            using (StreamWriter writer = File.CreateText(filepath))
            {
                await writer.WriteAsync(text);
            }
        }

        public async Task<string> ReadTextAsync(string filename)
        {
            string filepath = GetFilePath(filename);
            using (StreamReader reader = File.OpenText(filepath))
            {
                return await reader.ReadToEndAsync();
            }
        }

        string GetFilePath(string filename)
        {
            string docsPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
            return Path.Combine(docsPath, filename);
        }
    }
}
```

However, there is no asynchronous version of the `File.Exists` method, so a `Task<bool>` is simply constructed from the result using the static `FromResult` method.

The Android implementation of `FileHelper` is the same as the iOS version. The PCL project in **NoteTaker6** has a new static `FileHelper` method that caches the `IFileHelper` object and hides all the `DependencyService` calls:

```
using System.Threading.Tasks;
using Xamarin.Forms;

namespace NoteTaker6
{
    static class FileHelper
    {
        static IFileHelper fileHelper = DependencyService.Get<IFileHelper>();

        public static Task<bool> ExistsAsync(string filename)
        {
            return fileHelper.ExistsAsync(filename);
        }

        public static Task WriteTextAsync(string filename, string text)
        {
            return fileHelper.WriteTextAsync(filename, text);
        }

        public static Task<string> ReadTextAsync(string filename)
        {
            return fileHelper.ReadTextAsync(filename);
        }
    }
}
```

These methods simply return the same return values as the underlying asynchronous methods.

The `Note` class is mostly the same as before, but the `Save` and `Load` methods are now `SaveAsync` and `LoadAsync`:

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Threading.Tasks;

namespace NoteTaker6
{
    class Note : INotifyPropertyChanged
    {
        …

        public Task SaveAsync(string filename)
        {
            string text = this.Title + "\n" + this.Text;
            return FileHelper.WriteTextAsync(filename, text);
        }

        public async Task LoadAsync(string filename)
        {
            string text = await FileHelper.ReadTextAsync(filename);

            // Break string into Title and Text.
            int index = text.IndexOf('\n');
            this.Title = text.Substring(0, index);
            this.Text = text.Substring(index + 1);
        }

        …
    }
}
```

Neither of these methods returns a value, but because they are asynchronous, the methods return a `Task` object that can be awaited. There are a couple ways to deal with such methods. The `SaveAsync` method simply returns the return value from `FileHelper.WriteTextAsync`, which is also a `Task` object. The `LoadAsync` method has no `return` statement, although it could surely end with an empty `return` statement. The `SaveAsync` method could have an implicit empty `return` statement but the `WriteTextAsync` call would have to be preceded with `await`, and because of the existence of that `await` operator, the method would need an `async` modifier:

```
public async Task SaveAsync(string filename)
{
    string text = this.Title + "\n" + this.Text;
    await FileHelper.WriteTextAsync(filename, text);
}
```

These new function definitions require changes to the code in the `NoteTaker6Page` constructor as well, and you have a couple choices. You can define the `Clicked` handler for the **Load** button like so:

```
loadButton.Clicked += (sender, args) =>
    {
        note.LoadAsync(FILENAME);
    };
```

You'll get a warning from the compiler that you might consider using the `await` operator. But strictly speaking, you don't need to. What happens is that the `Clicked` handler calls `LoadAsync` but doesn't wait for it to complete. The `Clicked` handler returns back to the button that fired the event before the file has been loaded.

You can use `await` in a lambda function but you must precede the argument list with the `async` modifier:

```
loadButton.Clicked += async (sender, args) =>
    {
        await note.LoadAsync(FILENAME);
    };
```

For the **Save** button, you probably don't want to enable the **Load** button until the save operation has completed, so you'll want the `await` in there:

```
saveButton.Clicked += async (sender, args) =>
    {
        await note.SaveAsync(FILENAME);
        loadButton.IsEnabled = true;
    };
```

Actually, if you start thinking about asynchronous file I/O, you might start getting nervous—and justifiably so. For example, what if you press the **Save** button and, while the file is still in the process of being saved, you press the **Load** button? Will an exception result? Will you only load half the file because the other half hasn't been saved yet?

It's possible to avoid such problems on the user-interface level. If you want to prohibit a button press at a particular time, you can disable the button. Here's how the program can prevent the buttons from interfering with each other or with themselves:

```
saveButton.Clicked += async (sender, args) =>
    {
        saveButton.IsEnabled = false;
        loadButton.IsEnabled = false;
        await note.SaveAsync(FILENAME);
        saveButton.IsEnabled = true;
        loadButton.IsEnabled = true;
    };

loadButton.Clicked += async (sender, args) =>
    {
        saveButton.IsEnabled = false;
        loadButton.IsEnabled = false;
        await note.LoadAsync(FILENAME);
        saveButton.IsEnabled = true;
        loadButton.IsEnabled = true;
    };
```

It's unlikely that you'll run into problems with these very small files and solid-state memory, but it never hurts to be too careful.

As you'll recall, the **Load** button must be initially disabled if the file doesn't exist. The await operator is a full-fledged C# operator, so you should be able to do something like this:

```
Button loadButton = new Button
{
    Text = "Load",
    IsEnabled = await FileHelper.ExistsAsync(FILENAME),
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
```

Yes, this works!

And yet, it doesn't. The problem is that the method that contains this code must have an async modifier and the async modifier is not allowed on a constructor. But there's a way around this restriction. You can put all the initialization code in a method named Initialize with the async modifier and then call it from the constructor:

```
public NoteTaker6Page()
{
    Initialize();
}

async void Initialize()
{
    …
}
```

The Initialize method will execute up to the point of the await operator and then return back to the constructor, which will then return back to the code that instantiated the class. The remainder of the Initialize method continues after the ExistsAsync method returns a value and the IsEnabled property is set.

But in the general case, do you want a good chunk of your page initialization to be delayed until a single property is set that has no other effect on the page? Probably not.

Even with the availability of await, there are times when it makes sense to devote a completed handler to a chore. When an asynchronous method returns a Task object, the syntax is a little different for specifying a completed handler, but here it is:

```
FileHelper.ExistsAsync(FILENAME).ContinueWith((task) =>
    {
        loadButton.IsEnabled = task.Result;
    });
```

Here's the complete NoteTaker6Page class:

```
class NoteTaker6Page : ContentPage
{
    static readonly string FILENAME = "test.note";

    public NoteTaker6Page()
    {
```

```
// Create Entry and Editor views.
Entry entry = new Entry
{
    Placeholder = "Title (optional)"
};

Editor editor = new Editor
{
    Keyboard = Keyboard.Create(KeyboardFlags.All),
    BackgroundColor = Device.OnPlatform(Color.Default,
                                        Color.Default,
                                        Color.White),
    VerticalOptions = LayoutOptions.FillAndExpand
};

// Set data bindings.
Note note = new Note();
this.BindingContext = note;
entry.SetBinding(Entry.TextProperty, "Title");
editor.SetBinding(Editor.TextProperty, "Text");

// Create Save and Load buttons.
Button saveButton = new Button
{
    Text = "Save",
    HorizontalOptions = LayoutOptions.CenterAndExpand
};

Button loadButton = new Button
{
    Text = "Load",
    IsEnabled = false,
    HorizontalOptions = LayoutOptions.CenterAndExpand
};

// Set Clicked handlers.
saveButton.Clicked += async (sender, args) =>
    {
        saveButton.IsEnabled = false;
        loadButton.IsEnabled = false;
        await note.SaveAsync(FILENAME);
        saveButton.IsEnabled = true;
        loadButton.IsEnabled = true;
    };

loadButton.Clicked += async (sender, args) =>
    {
        saveButton.IsEnabled = false;
        loadButton.IsEnabled = false;
        await note.LoadAsync(FILENAME);
        saveButton.IsEnabled = true;
        loadButton.IsEnabled = true;
    };
```

```
        // Check if the file is available.
        FileHelper.ExistsAsync(FILENAME).ContinueWith((task) =>
            {
                loadButton.IsEnabled = task.Result;
            });

        // Assemble page.
        this.Padding = new Thickness(10, Device.OnPlatform(20, 0, 0), 10, 0);

        this.Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "Title:"
                },
                entry,
                new Label
                {
                    Text = "Note:"
                },
                editor,
                new StackLayout
                {
                    Orientation = StackOrientation.Horizontal,
                    Children =
                    {
                        saveButton,
                        loadButton
                    }
                }
            }
        };
    }
}
```

With no error handling, the program is implicitly assuming that there will be no problems encountered when loading and saving files. Error handling can be implemented by enclosing any asynchronous method call with `await` in a `try` and `catch` block. If you want to deal with errors “silently” without informing the user, you can implement this check in the `Note` class. If you need to display an alert to the user, it makes more sense to perform the check in the `ContentPage` derivative.

What’s next?

Much of the infrastructure is in place for storing and retrieving a single note. We are now ready to save and retrieve multiple notes and display them in a scrollable list for reference or editing.